Федеральное государственное бюджетное образовательное учреждение высшего образования «Новосибирский государственный технический университет»

На правах рукописи

Сердюков Константин Евгеньевич

РАЗРАБОТКА СИСТЕМ ИНТЕЛЛЕКТУАЛЬНОЙ ПОДДЕРЖКИ АНАЛИЗА И ТЕСТИРОВАНИЯ ПРОГРАММ

Специальность 05.13.11 – Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей (технические науки)

Диссертация на соискание ученой степени
кандидата технических наук

Научный руководитель:
Авдеенко Татьяна Владимировна
доктор технических наук., профессор

Новосибирск – 2022

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Актуальность и степень разработанности темы исследования. В настоящее время уровень развития интеллектуальных технологий позволяет говорить о возможности их применения в наиболее сложных, инновационных сферах, к которым, несомненно, относится программная инженерия, занимающаяся созданием, внедрением и сопровождением современного программного обеспечения (ПО). Классический жизненный цикл ПО включает в себя такие этапы, как определение и анализ требований, проектирование, программирование, тестирование и отладка, эксплуатация и сопровождение. Для обеспечения высокого качества разрабатываемого ПО большое значение имеют методы верификации. Они применяются на всех этапах жизненного цикла и позволяют сделать вывод о качестве разрабатываемого ПО на основе проверки соответствия между программой и заявленными к ней требованиями. Верификация является важным процессом разработки, а тестирование, относящееся к методам динамической верификации, выделяется в отдельный этап жизненного цикла ПО.

Этап тестирования, направленный на проверку соответствия между ожидаемыми результатами и реальным поведением программы на специально подобранном наборе тестов (тестовых данных), является одним из самых дорогостоящих и трудозатратных, и может занимать до 40–60% от общего времени создания ПО. Поэтому разработка моделей и алгоритмов интеллектуальной поддержки этапа тестирования ПО является актуальной задачей.

Тестированием программного обеспечения, в том числе системного, а также разработкой тестов, занимались такие ученые как Бурдонов И.Б., Денисова А.Л, Панков Д.А., Мутилин В.С. Новые методы верификации предикатных и автоматных программ, и средств их визуализации, предложены в работах Шелехова В.И., Непомнящего В.А., Зюбина В.Е. Вопросами тестирования и верификации программных средств при разработке автоматизированных систем

управления в области спецхимии занимались Попов Ф.А., Кащеева Е.В.; в области добычи угля – Окольнишников В.В; в аэрокосмической области – Тюгашев А.А.

Генерация тестовых данных – сложный и трудоемкий процесс. Его автоматизация, хотя бы частичная, является актуальной исследовательской задачей, решение которой могло бы повысить эффективность тестирования ПО. Анализ существующих исследований, методов и подходов в области применения методов автоматической генерации тестовых данных показал, что в настоящее время преимущественно используемым в производстве программного обеспечения подходом является применение слепой стратегии случайной генерации тестовых данных. В то же время, анализ проводимых научных исследований показывает, что существуют подходы, разработка и применение которых может значительно улучшить качество генерируемых тестов, выражаемое в степени покрытия ими тестируемого кода (термин «покрытие кода» употребляется в соответствии с [52], и означает прохождение вычислений тестируемой программы, инициированное множеством тестовых наборов, по максимально возможному числу путей).

Среди таких продвинутых подходов к генерации тестовых данных исторически первыми появились статические методы символьного анализа кода программы. Генерация тестовых данных в результате такого анализа сводилась к автоматическому формированию и разрешению в символьном виде системы уравнений и неравенств, получающихся логическим объединением и пересечением всех условий в тестируемой программе. Несомненным достоинством статического подхода является получение результатов в символьном виде, что дает возможность аналитически определять подобласти значений тестовых наборов, которые гарантируют проход вычислений по заданным частям кода.

Однако существенным ограничением возможности применения статического подхода является проблема вычислительной сложности символьных вычислений даже для задач относительно небольшой размерности. Поэтому в настоящее время более реалистичным и эффективным для практического использования в

компаниях по производству ПО является динамический подход, основанный на фактическом выполнении тестируемой программы при сгенерированных специальным образом значениях входных переменных (наборе тестов) и последующем анализе потоков данных. Развитием этого направления занимались Давыдов А.А., Demillo R., Gelrich R., Nikravan E., Parsa S. и т.д.

Наиболее перспективными методами реализации динамического подхода к генерации тестовых данных являются эволюционные методы оптимизации. Эволюционная парадигма, которая лежит в основе генетического алгоритма (ГА), использует множество случайных тестовых данных, сгенерированных на начальном этапе, после чего проводится последовательная «эволюция» данных с целью улучшения качества покрытия тестируемого кода. В связи с вышеизложенным возникает предположение о возможности адаптации генетического алгоритма для реализации идеи эволюционного улучшения тестовых данных с точки зрения максимизации покрытия ими тестируемого кода.

Вопросы применения генетического алгоритма к генерации тестовых данных рассматриваются в работах Anusha M., Berndt D., Girdis M. Maragathavalli P., Praveen R., Watkins A. и др. Однако существующие исследования сосредоточены на решении локальных проблем, например, на нахождении конкретного набора данных, покрывающего отдельные заданные операторы. В то же время для решения практической задачи всестороннего тестирования ПО актуальна разработка методов генерации множества наборов, обеспечивающих наиболее полное покрытие всего тестируемого кода, с учетом его многосвязной сложнологической структуры, включая рекурсию.

Цель работы, таким образом, заключается в разработке и исследовании методов автоматической генерации тестовых данных на основе модификаций генетического алгоритма для наиболее полного покрытия программного кода.

Для достижения заданной цели поставлены и решены следующие **задачи**:

1. Провести анализ существующих исследований, методов и подходов в области применения методов автоматической генерации тестовых данных;

2. Разработать алгоритм генерации набора тестовых данных для покрытия наиболее сложного пути тестируемого кода на основе метрик оценки сложности.

3. Модифицировать разработанный алгоритм для получения множества наборов данных, обеспечивающих наиболее полное покрытие кода;

4. Исследовать различные варианты функции приспособленности ГА для обеспечения наибольшего покрытия за счёт большего разнообразия множества сгенерированных тестовых наборов;

5. Реализовать приложение для генерации множества наборов тестовых данных с использованием предложенного алгоритма и его модификаций.

Область исследования. Диссертация соответствует области исследования п.1 «Модели, методы и алгоритмы проектирования и анализа программ и программных систем, их эквивалентных преобразований, верификации и тестирования» паспорта специальности 05.13.11 – «Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей».

Объектом исследования является процесс генерации тестовых данных как основа этапа тестирования при разработке ПО.

Предметом исследования являются модели, методы и алгоритмы интеллектуальной поддержки анализа и тестирования ПО

Теоретическая значимость проведённого исследования заключается в развитии эволюционного подхода для решения задачи автоматической генерации тестовых данных для наиболее полного покрытия кода за счет формулирования специального вида функции приспособленности, учитывающей не только сложность пути, но и разнообразие множества тестовых наборов.

Практическая значимость. Результаты диссертационного исследования могут использоваться в компаниях, занимающихся разработкой ПО, для

автоматической генерации тестовых данных с целью повышения качества тестирования.

Научная новизна исследования заключается в следующем:

1. Осуществлена формальная постановка задачи генерации тестовых данных для ее решения с помощью генетического алгоритма, включающая математический вид функции приспособленности на основе метрик оценки сложности кода;

2. Сформулированы две новые модификации функции приспособленности, направленные на увеличение степени покрытия тестируемого кода. Первая модификация заключается во введении дополнительной аддитивной компоненты в функцию приспособленности, отвечающей за разнообразие популяции. Во второй модификации разнообразие достигается за счет динамического изменения весов операторов в зависимости от степени их покрытия в предшествующем поколении;

3. Разработаны эвристические алгоритмы генерации тестовых данных на основе предложенных формальных эволюционных постановок задач и различных вариантов функции приспособленности;

4. Разработано программное приложение, реализующее предложенные методы генерации тестовых наборов с модифицированной функцией приспособленности для обеспечения максимального покрытия тестируемого кода с минимально необходимым количеством наборов.

Методы исследования. Основой методологии представленного исследования является теория программной инженерии, модели, методы и алгоритмы тестирования ПО, методы оптимизации, генетический алгоритм.

Основные положения, выносимые на защиту:

1. Формальная постановка задачи генерации тестовых данных и математический вид функции приспособленности на основе метрик сложности кода;

2. Алгоритмы генерации данных на основе многократного запуска и алгоритма с дополнительным параметром, отвечающим за разнообразие;

3. Модификации функции приспособленности, обеспечивающие большее разнообразие тестовых наборов;

4. Программное приложение, реализующее предложенные методы и подходы.

Степень достоверности результатов работы. Достоверность полученных результатов определяется использованием современного научно-методического аппарата, а также взаимной согласованностью результатов, полученных для различных сочетаний параметров и модификаций разработанных алгоритмов.

Представленные в работе научные результаты получены в рамках выполнения работ по грантам – Грант Министерства образования и науки РФ в рамках проектной части государственного задания, проект № 2.2327.2017/4.6 «Интеграция моделей представления знаний на основе интеллектуального анализа больших данных для поддержки принятия решений в области программной инженерии» (2017-2019 г.); Грант Российского фонда фундаментальных исследований в рамках выполнения научного проекта № 19-37-90156 (Аспиранты) «Разработка и исследование методов интеллектуального анализа и тестирования программного кода» (2019-2021 г.); Грант Министерства Науки и Высшего Образования Госзадания проект № FSUN-2020-0009 «Моделирование системной организации когнитивных функций с применением интеллектуального анализа массивов психометрических и нейрофизиологических данных» (2020-2022 г.).

Предлагаемые автором методы и алгоритмы генерации тестовых данных на основе эволюционных алгоритмов используются при разработке программного обеспечения в ООО «Дежавю». Результаты работы используются в учебном процессе Новосибирского государственного технического университета в рамках дисциплин «Интеллектуальные информационные системы», «Интеллектуальные системы и технологии», «Программная инженерия», «Методы оптимизации».

Апробация работы. Основные результаты работы были представлены на конференциях: 12th International Conference on Advances in Swarm Intelligence

(ICSI’21) (Qingdao, China, 2021); 14th International Symposium “Intelligent Systems – 2020” (INTELS’20) (г. Москва, 2020); 12th International Conference “Data Analytics and Management in Data Intensive Domains” (DAMDID) (г. Воронеж, 2020); междунар. конф. и молодеж. шк. «Информационные технологии и нанотехнологии» (г. Самара, 2017, 2018, 2019, 2020, 2021); Раб. семинар в рамках 12 междунар. Ершовской конф. по информатике (PSI'19) (г. Новосибирск, 2019); всерос. науч.-техн. конф. студентов, аспирантов и молодых ученых с междунар. участием «Измерения, автоматизация и моделирование в промышленности и научных исследованиях» (г. Бийск, 2018, 2019); всерос. науч. конф. молодых ученых «Наука. Технологии. Инновации» (г. Новосибирск, 2016, 2017).

Публикации. По теме диссертационной работы опубликовано 23 работы, в том числе 2 работы опубликованы в научных журналах из перечня ВАК [1, 2], 10 публикаций – в изданиях, индексируемых Web of Science и Scopus [3-12], 2 свидетельства о регистрации программы для ЭВМ [13, 14], 9 публикаций в сборниках трудов международных и российских конференций [15-23].

Личный вклад автора. Результаты научных исследований, представленных в диссертационной работе, были получены при непосредственном участии соискателя, которое заключалось в разработке и реализации алгоритмов генерации тестовых данных, постановке вычислительных экспериментов и апробации полученных результатов. Доля личного вклада в публикациях, выполненных в соавторстве, составляет не менее 50%.

Структура и объем работы. В диссертации представлено введение, четыре главы по тематике исследования, заключение, список литературы, в котором содержится 110 наименований, и 4 приложения. Полный объём работы составляет 166 страниц, включая 10 таблиц и 50 рисунков.

ГЛАВА 1 ГЕНЕРАЦИЯ ТЕСТОВЫХ ДАННЫХ ДЛЯ ЭТАПА ТЕСТИРОВАНИЯ ПРИ РАЗРАБОТКЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В первой главе рассмотрены основные модели и этапы жизненного цикла разработки ПО, представлены базовые подходы к тестированию как этапу разработки ПО. Приводятся основные понятия и определения в области генерации тестовых данных, проводится анализ существующих исследований в данной области, формулируется основная цель исследования.

1.1 Жизненный цикл разработки программного обеспечения

Разработка современного программного обеспечения в настоящее время проводится в соответствии с принципами программной инженерии. Словарь терминов системной и программной инженерии ISO/IEC/IEEE (SEVOCAB) [24] определяет программную инженерию как «применение систематического, упорядоченного, поддающегося количественной оценке подхода к разработке, эксплуатации и сопровождению программного обеспечения, то есть применение инженерии к программному обеспечению».

Таким образом, программная инженерия – это инженерная дисциплина, которая занимается всеми аспектами разработки ПО от ранних стадий определения требований к системе до сопровождения разработанного ПО после того, как оно было введено в эксплуатацию. Программная инженерия не только рассматривает непосредственно технические процессы разработки ПО, но также включает в себя такие виды деятельности, как управление проектами и применение инструментария, методов и теорий для поддержки производства ПО [25, 26, 27, 28]. В долгосрочной перспективе обычно дешевле использовать методы и приемы программной инженерии, чем просто писать программы без применения каких-либо специализированных методов.

В программной инженерии выделяют понятие жизненного цикла ПО, под которым понимают специфическую для проекта последовательность этапов,

определяющих необходимые действия от разработки концепта до выведения ПО из эксплуатации [29].

1.1.1 Модели жизненного цикла

Жизненный цикл разработки системы состоит из ряда четко определенных этапов, которые применяются разработчиками систем. Использование методологии жизненного цикла определяется необходимостью производства высококачественных систем, которые соответствуют (или превосходят) ожиданиям клиентов – путем прохождения каждого четко определенного этапа в запланированные сроки и с установленными затратами [30]. Современные компьютерные системы сложны, и часто получаются объединением нескольких более простых систем, потенциально предоставляемых различными поставщиками программного обеспечения. Для управления разработкой сложного ПО был создан ряд моделей, таких как каскадная, спиральная, итеративная, гибкая разработка программного обеспечения, V модель и другие.

Каскадная модель

Исторически первой была предложена каскадная модель жизненного цикла ПО. Её суть состоит в разделении на линейные последовательные этапы, где каждый из этапов зависит от результатов предыдущего и жестко ограничен собственной спецификацией (Рисунок 1.1).

Преимущество каскадной модели заключается в том, что она позволяет наглядно распределить основные работы по этапам и обеспечить более эффективный контроль за разработкой. Можно установить график работ с указанием крайних сроков для каждого этапа разработки, когда разрабатываемый продукт будет проходить этапы модели процесса разработки один за другим. Недостатком каскадной модели является недостаточная гибкость, она не позволяет пересматривать отдельные элементы программы после начала разработки. Например, если программа находится на стадии тестирования, очень трудно

вернуться и внести определенные изменения в код, если это не было хорошо задокументировано.

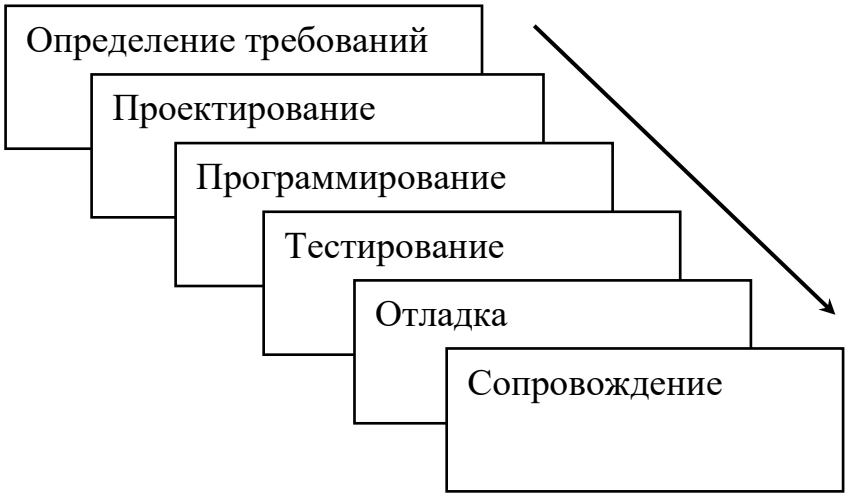


Рисунок 1.1 – Каскадная модель жизненного цикла программного обеспечения

Итеративная модель

Итеративная модель начинается с определения требований к программному обеспечению и итеративно улучшает разрабатываемое ПО (разные версии), пока не будет окончательно доработана вся система. На каждой итерации вносятся изменения в дизайн, и добавляются новые функциональные возможности (Рисунок 1.2). Основная идея этого метода заключается в разработке системы посредством повторяющихся циклов (итеративно), и небольшими частями за раз (инкрементально).

Особенностью использования итеративного жизненного цикла разработки ПО является тщательная проверка требований, а также тестирование и последующая отладка каждой версии программного обеспечения в рамках каждого цикла (итерации) модели. По мере того, как программное обеспечение развивается через последовательные итерации, использование тестовых наборов необходимо повторять и расширять для проверки каждой версии программного обеспечения.

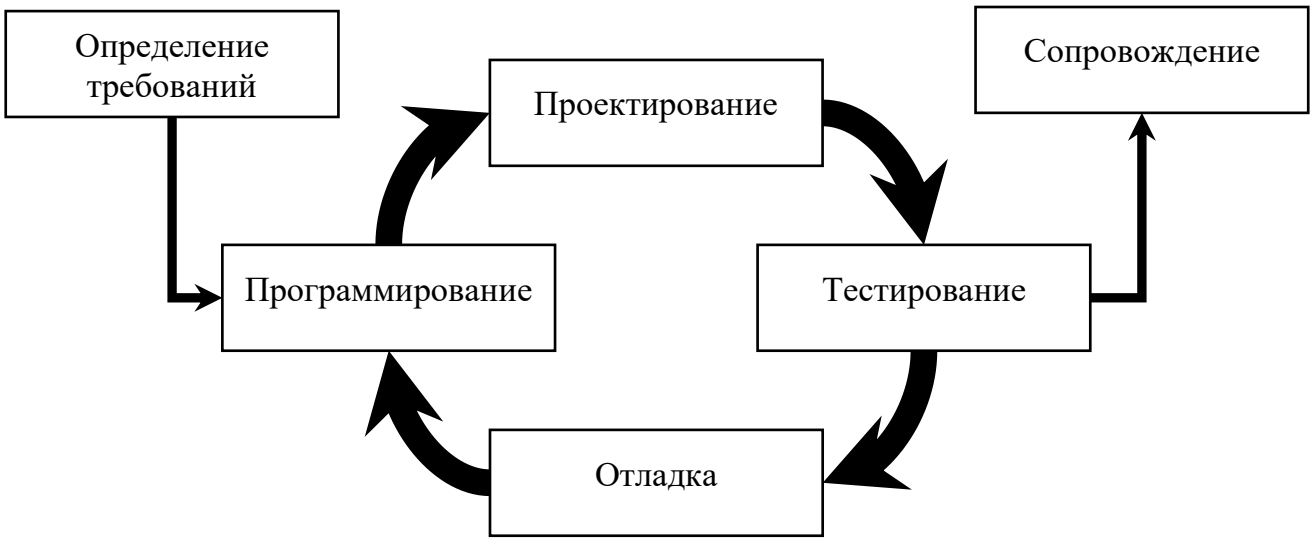


Рисунок 1.2 – Итеративная модель жизненного цикла

Преимущество данной модели заключается в том, что уже на самых первых итерациях разработки имеется рабочая модель системы, что упрощает поиск функциональных или конструктивных недостатков. Обнаружение проблем на ранней стадии разработки позволяет принимать корректирующие меры в ограниченном бюджете. Недостатком этой модели является то, что она применима только к крупным и громоздким проектам разработки программного обеспечения. Это связано с тем, что небольшую программную систему трудно разбить на дополнительные, отдельно обслуживаемые, модули.

Спиральная модель жизненного цикла

Спиральная модель сочетает в себе идею итеративной модели с систематическими, управляемыми аспектами каскадной модели. Спиральная модель представляет собой комбинацию модели итеративного процесса разработки и модели последовательной линейной разработки, то есть каскадной модели с очень большим упором на анализ рисков. Это позволяет производить инкрементные выпуски продукта или инкрементное уточнение на каждой итерации по спирали.

Преимуществом спиральной модели жизненного цикла является возможность добавлять модули для разрабатываемого ПО, когда они становятся доступными или известными. Это гарантирует отсутствие противоречий с предыдущими требованиями и дизайном.

Данный метод совместим с подходами, предусматривающими несколько версий и прототипов программного обеспечения, что позволяет упорядоченно переходить к этапу сопровождения ПО. Еще одним положительным аспектом этого метода является то, что спиральная модель вынуждает пользователей на раннем этапе участвовать в разработке системы, что позволяет уточнять требования к разработке и дополнять спецификации. С другой стороны, для завершения таких продуктов требуется очень строгое управление, и есть риск запустить спираль в бесконечный цикл, когда с каждой новой версией программы требования будут изменяться, или расти.

Гибкая (Agile) модель жизненного цикла

Гибкая модель, или модель Agile, представляет собой комбинацию итеративных моделей жизненного цикла с акцентом на адаптируемость процесса и удовлетворение потребностей клиентов за счет быстрого предоставления работающего программного продукта. Гибкие методы разбивают продукт на небольшие отдельные сборки, которые формируются в итерациях. Каждая итерация обычно длится от одной до трех недель, и в конце итерации заказчику и важным заинтересованным сторонам демонстрируется рабочий продукт.

В гибкой модели считается, что каждый проект нужно обрабатывать по-разному, и существующие методы должны быть адаптированы к требованиям проекта, а не проект должен быть подстроен под модель. Задачи разделяются на подзадачи, соответствующие определённым функциям, которые должны быть выполнены в определённые временные рамки. Применяется итеративный подход, в котором после каждой итерации предоставляется рабочая сборка программного обеспечения. Каждая сборка является инкрементальной с точки зрения числа

функций, т.е. в каждой последующей сборке больше функций, чем в предыдущей. Окончательная версия включает все функции, требуемые заказчиком.

1.1.2 Основные этапы жизненного цикла

В п. 1.1.1 были представлены модели жизненного цикла разработки, включающие основные этапы разработки в контексте проекта. Несмотря на существенные различия, набор этапов в каждой методологии одинаков [31]. Рассмотрим каждый из этапов более подробно.

Определение требований

Первым шагом процесса программной инженерии является сбор, анализ и документирование требований к разрабатываемому продукту. Требования должны быть понятны, однозначно определяемы, исчерпывающи, полны, но при этом кратки. Анализ требований должен прояснять и определять функциональные требования и ограничения проекта. Функциональные требования определяют количество, качество, покрытие, сроки и доступность конечного ПО. Ограничения проекта определяют факторы, ограничивающие гибкость проекта, такие как: условия или ограничения программной среды, защита от внутренних или внешних угроз, договорные, клиентские или нормативные стандарты.

Существенным свойством всех требований к программному обеспечению является то, что они поддаются проверке. Для функциональных требований это проверка отдельных функций, для нефункциональных требований – проверка на системном уровне. Требования к программному обеспечению на этапе тестирования должны гарантировать, что они могут быть проверены в рамках имеющихся ограниченных ресурсов. Как правило, требования к программному обеспечению идентифицируются однозначно, так что они могут подвергаться управлению конфигурацией в течение всего жизненного цикла ПО.

Анализ требований включает определение потребностей и целей потребителя в контексте планируемого использования, среды использования и определенных характеристик системы для определения требований к функциям системы.

Проектирование программного обеспечения

После того, как требования были получены и описаны, необходимо спроектировать будущее программное обеспечение. В международном стандарте ISO/IEC/IEEE [24] проектирование программного обеспечения определяется как «процесс определения программной архитектуры, компонентов, модулей, интерфейсов и данных в программной системе для соответствия определенным требованиям».

Если рассматривать проектирование программного обеспечения как процесс, то его можно определить, как деятельность в течение жизненного цикла программной инженерии, в ходе которой требования к программному обеспечению анализируются с целью описания внутренней структуры ПО, являющееся основой для его разработки [32]. Проект описывает архитектуру программного обеспечения, то есть, из каких компонентов оно состоит, и как они организованы, а также интерфейсы между ними. Компоненты также должны описываться на уровне детализации, что позволит в дальнейшем закодировать их. Дизайн программного обеспечения играет важную роль в его разработке – происходит создание различных моделей, которые в совокупности образуют своего рода модель разработки. Можно проанализировать и оценить эти модели, чтобы определить, позволяют ли они обеспечить выполнение различных требований. Также могут быть предложены и оценены альтернативные решения и компромиссы, если выполнение некоторых требований затруднительно. Наконец, можно использовать полученные модели для планирования последующих действий, таких как тестирование и валидация системы. Они могут служить как основой для создания тестовых данных, так и использоваться в процессе проверки уже разработанного ПО.

Программирование

Программирование является процессом написания исходного кода программы. Часто в данный этап включают и юнит-тестирование (модульное тестирование). Фактически, под этапом программирования можно понимать преобразование требований, определённых на этапе проектирования, в рабочую программу. Написание программного обеспечения обычно создает наибольшее количество элементов конфигурации, которыми необходимо управлять в программном проекте (исходные файлы, документация, тестовые наборы и т.д.).

Программирование непосредственно связано с проектированием и тестированием, поскольку для процесса написания программы часто необходимо наличие входных и выходных данных. Входные данные для программирования являются результатом проектирования, а выходные используются для тестирования.

Границы между проектированием, программированием и тестированием могут существенно различаться для каждого ПО в зависимости от процессов жизненного цикла, которые определяются в проекте. Также стоит отметить, что первоначальное проектирование ПО часто не включает детального описания будущей программы, хотя определенные детали все же описываются. Значительная часть проектирования происходит во время написания кода, когда уже можно определить точные связи между разными модулями. Таким образом, этап программирования тесно связан с этапом проектирования.

На протяжении всего процесса программирования разрабатываемая программа постоянно тестируется, т.е. тестирование проводится для отдельных написанных модулей и функций. Таким образом, программирование очень тесно связано с последующим тестированием.

Тестирование

Тестирование – это проверка разработанной программы на соответствие исходным требованиям, спецификациям, полученным на этапе анализа проекта

разрабатываемого ПО. Фактически, тестирование – это деятельность, при которой система или её компонент запускается в определенных условиях при определенных входных данных, результаты наблюдаются или регистрируются, после чего выполняется оценка протестированного фрагмента, или аспекта, системы или компонента на предмет соответствия требованиям [33].

Отладка (дебаггинг) кода

Дебаггинг (Debugging), также называемый отладкой, является процессом нахождения, анализа и исправления причин возникновения ошибок в коде.

Отладка проводится итерационным методом, в котором программа останавливается в определённом месте на определенном операторе (точка останова) для отображения и установки значений переменных. Это позволяет исследовать текущее состояние системы и найти соответствующие этому состоянию несоответствия [34].

В отличие от тестирования, для процесса отладки характерно изменение написанного кода для локализации места возникновения ошибок. Это возможно либо за счет применения более простых наборов данных, которые позволяют выходить на заданные пути программного кода, либо за счет проверки ограниченной части кода, например, определённого модуля или функции. Таким образом, тестирование сосредоточено на поиске ошибок [35], сбоев и т.д., тогда как этап отладки начинается уже после обнаружения ошибки [36].

Сопровождение разработанного программного обеспечения

В результате разработки программного обеспечения создается программный продукт, удовлетворяющий требованиям конечного потребителя. Соответственно, программный продукт должен меняться или развиваться по мере своего создания и после него. После ввода в эксплуатацию может меняться среда применения, могут обнаруживаться ошибки или сбои, появляться новые требования пользователей, которые не были учтены на этапе определения требований [37]. Этап сопровождения в жизненном цикле начинается во время периода

гарантийного обслуживания или после предоставления готового ПО, но процесс технического сопровождения начинается намного раньше.

По истечении срока полезного использования программного обеспечения его необходимо вывести из эксплуатации. Для принятия решения о прекращении использования, проводится детальный анализ, который охватывает требования к выводу, влияние на текущие бизнес-процессы, планируемую замену, график и затраты. Также может быть включена необходимость обеспечить доступность архивных копий данных.

1.2 Оценка качества программного обеспечения

С увеличением размера и сложности разрабатываемой программы увеличивается вероятность допустить ошибку или не учесть определённое требование. Поэтому для обеспечения высокого качества конечного продукта большое значение имеют методы верификации, применяющиеся на всех этапах жизненного цикла разработки ПО и позволяющие выявлять ошибки для последующего их устранения.

1.2.1 Верификация программного обеспечения

Верификация является процессом проверки соответствия между разрабатываемым ПО и заявленными к нему требованиями. Она применяется для обнаружения ошибок, несоответствий, дефектов, некорректно описанных и реализованных требований.

Верификация должна выполняться на протяжении всего жизненного цикла разработки программного обеспечения для проверки конечных и промежуточных результатов, оценка которых позволяет сделать вывод о качестве разработанной программы. Оценка качества используется для планирования следующих этапов разработки, принятии решений о корректировке требований, окончании разработки или о передаче готового ПО заказчику.

Существуют различные методы верификации, которые можно разделить на следующие группы [38]:

- Эмпирические. Включают в себя методы экспертизы;
- Формальные. Используют математический аппарат при проведении верификации;
- Динамические. Проверяют работу программы при непосредственном запуске.

Экспертиза позволяет проверить разработанный код и документацию на соответствие нормам и стандартам, принятым в организации, отрасли или стране. Она может быть проведена на любом из этапов жизненного цикла и позволяет обнаружить практически любые виды ошибок, но её невозможно автоматизировать, и для её проведения необходимо постоянное участие специалиста или группы специалистов в процессе разработки.

Методы формальной верификации основаны на использовании математической модели программы для проведения верификации, без необходимости обращения к реализации [39]. Требования переводятся в вид спецификаций, и они проверяются на соответствие математической модели. Модель может быть представлена в различных нотациях. В [40, 41], например, используются онтологии для описания верифицируемой системы и требований к ней. Недостатками формальных методов верификации является ограниченный круг решаемых задач и сложность построения математических моделей [42, 43].

Методы формальной верификации чаще всего относятся к статическим методам. Для выполнения статической верификации необходим только исходный код программы, реальный запуск программы не производится. В отличие от динамических методов, статический анализ позволяет проверить все возможные пути выполнения программы. Наибольшее распространение получили две группы статических методов – методы дедуктивного анализа программ и методы проверки моделей [44].

В методах дедуктивного анализа для определения качества разработанной программы используется спецификации, которые обычно задаются в виде пред- и постусловий [45, 46]. Для методов дедуктивного анализ необходимо вручную устанавливать аннотации функций и циклов программы, что существенно осложняет применение данных методов, в особенности для верификации больших программ [47]. Тем не менее, они служат достаточно эффективным методом доказательства соответствия требованиям, поэтому могут применяться при проверке критически важных компонентов ПО.

В основе методов проверки моделей лежит построение модели, описывающей код проверяемой программы в явном или неявном виде. Методы проверки моделей для проведения анализа могут использовать граф потоков управления (п. 1.3.1) [44]. Проверяемые ограничения в методах проверки моделей направлены, чаще всего, не на полное описание требований, а на проверку частных требований или проверку отсутствия определённого класса ошибок.

В целом, статические методы верификации позволяют обеспечить довольно глубокий анализ ПО, но из-за высоких требований к реализации, их использование довольно ограничено [38]. Ввиду того, что в статических методах верификации не производится запуск программы, класс ошибок, которые можно обнаружить с их помощью, также ограничен. Из-за высокой сложности реализации статические методы применяются или для проверки критически важных компонентов ПО, или для анализа аппаратных и системных программ.

Для динамических методов, в отличие от статических, характерно выполнение программы при проведении анализа, то есть код не только должен быть написан, но и успешно скомпилирован. Соответственно, методы динамической верификации используются на уже разработанной программе или отдельном готовом компоненте, для оценки которых необходимо иметь реальные результаты работы. По этой причине, динамические методы не могут быть применены на начальных этапах разработки, необходимо иметь работающую

программу, её компоненты или, хотя бы, прототипы. Однако именно с использованием динамических методов можно проанализировать программу в реальной среде выполнения, чего сложно добиться другими методами верификации, и что позволяет обеспечить нахождение ошибок, которые проявляются только при непосредственном запуске программы.

К динамическим методам относят мониторинг и тестирование. При проведении мониторинга производится только наблюдение и оценка характеристик проверяемой программы. Мониторинг используется для определения данных о работе программы с использованием специального инструментария. В целом, мониторинг довольно является довольно ограниченным методом динамической верификации, более полным и эффективным методом является тестирование, который даже выделяют в отдельный этап жизненного цикла разработки ПО.

1.2.2 Тестирование программного обеспечения

Тестирование является методом динамической верификации, в рамках которого результаты работы тестируемой программы или её отдельной компоненты проверяются на соответствие проектным решениям, требованиям, общим задачам проекта, в рамках которого эта система разрабатывается или сопровождается.

В общем, тестирование состоит из проверки соответствия ожидаемых результатов и реального поведения программы на определенном наборе тестовых данных, на основе которого делается вывод о качестве написанного кода. Эффективность применения методов динамической верификации, в том числе и тестирования, напрямую зависит от количества и качества тестовых данных [48].

Тестирование всегда подразумевает выполнение тестируемой программы (Software-under-test, SUT) на входных данных. На практике, полный набор тестовых данных обычно можно считать бесконечно большим, а тестирование

проводится лишь на подмножестве всех возможных тестовых наборов, которое определяется различными критериями. Тестирование всегда подразумевает компромисс между ограниченными ресурсами, с одной стороны, и неограниченными требованиями к программному обеспечению, с другой.

Наблюдаемое поведение программы может быть проверено на соответствие потребностям пользователя (обычно называемое тестированием для валидации), на соответствие спецификациям (тестирование для верификации) или, возможно, на соответствие ожидаемому поведению, следующему из неявных требований.

Тестирование, как и верификация, должно проводиться на протяжении всего жизненного цикла разработки и сопровождения ПО. Действительно, планирование тестирования должно начинаться с ранних стадий процесса определения требований к программному обеспечению, а планы и методы тестирования должны систематически и непрерывно разрабатываться и, возможно, улучшаться и уточняться по мере разработки и верификации. Эти действия по планированию и проектированию тестирования предоставляют полезную информацию для разработчиков и помогают выявить потенциально слабые места, такие как противоречия при проектировании или двусмысленность в документации.

Подготовка тестов на ранних этапах жизненного цикла может помочь в обнаружении ошибок и несоответствий при определении требований, поэтому часто при разработке тестов также проводится и экспертиза артефактов верификации, которые служат основой для тестирования. Таким образом, тестирование можно рассматривать как средство предоставления информации о функциональных возможностях и характеристиках качества программного обеспечения, а также для выявления ошибок в тех случаях, когда появление ошибок не удалось предотвратить.

Для применения динамических методов верификации обычно требуется дополнительная подготовка — создание тестов, разработка тестовой системы,

позволяющей их выполнять или системы мониторинга, позволяющей контролировать определенные характеристики поведения проверяемой системы.

1.2.3 Основные подходы к тестированию

Тестирование относят к динамическим методам верификации (динамическое тестирование), который позволяет проверить программу в процессе её выполнения. Динамическое тестирование проводится после того, как программа достигла определенной стадии разработки и, как было сказано выше, оно выполняется в реальной среде выполнения. На вход подаются сгенерированные тестовые наборы, после чего можно сопоставить полученные результаты с планируемыми, описанными на этапах определения требований и проектирования, и сделать вывод о качестве разработанной программы.

Тем не менее, тестирование также может применяться в качестве статического подхода к верификации (статическое тестирование) [49, 50, 51], где при подготовке тестов производится анализ источников, обзор документов спецификации и требований, а также документов с описанием плана разрабатываемой программы.

Решение задачи генерации тестовых данных подразумевает доступ к исходному коду тестируемой программы, поэтому может быть выполнена только при динамическом тестировании, проводится с использованием подходов к тестированию и разработке тестов. Данные подходы используются для четкого определения цели тестирования программы, её отдельного компонента/модуля, или проекта в целом. Необходимо заранее планировать применение подходов к тестированию, потому что, например, тестирование только функциональности определённых компонентов системы может быть недостаточно для достижения общих целей тестирования и верификации. Определение конкретной цели тестирования и, следовательно, выбор соответствующего подхода к разработке

тестов, позволяет лучше концентрировать усилия для получения более качественных результатов.

Существующие подходы к тестированию можно условно разделить на две группы: функциональное тестирование и структурное тестирование. Далее рассмотрим эти подходы подробнее.

Функциональное тестирование

При функциональном тестировании внутренняя структура и конструкция тестовой программы (SUT) неизвестны или не рассматриваются, поэтому его часто называют тестированием черного ящика (Рисунок 1.3). Данный подход называется функциональным потому, что главной его целью является сопоставление входных данных и результатов выполнения, т.е. проверка работы функций. Другое применимое название к данному методу – тестирование на основе спецификаций. Связано это с тем, что единственным источником для проверки качества ПО являются спецификации требований, с которыми сравниваются результаты.

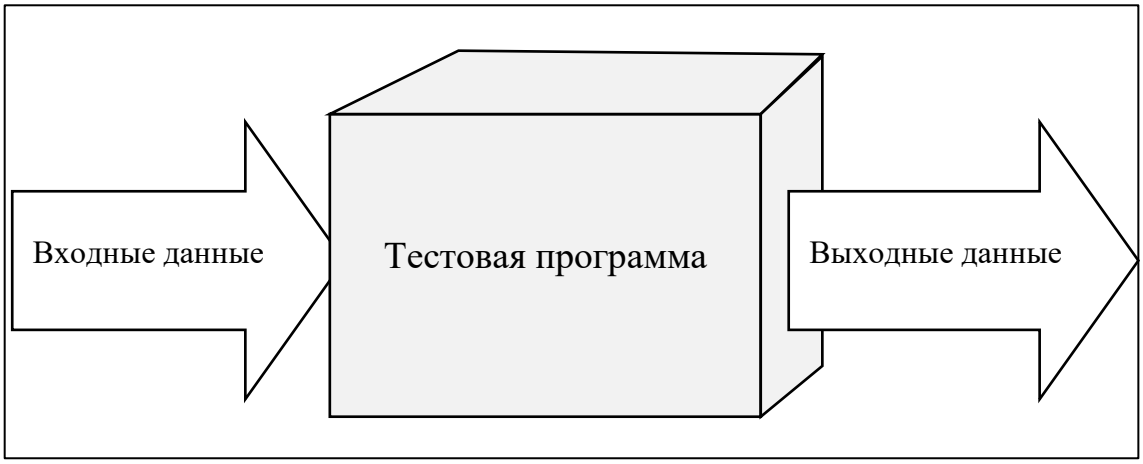


Рисунок 1.3 – Программа в виде черного ящика. Тестирование проводится с доступом только к входным и выходным данным, внутренняя структура программы неизвестна

Тестовые наборы для проверки программы определяются на основе применения методов верификации или предоставляются после применения альтернативного подхода (белого ящика). Тест со всеми возможными

комбинациями входных данных был бы полным тестом (то есть было бы возможно проверить полную функциональность), однако это нереалистично из-за огромного количества комбинаций значений выходных данных, что говорит об ограниченных возможностях данного метода для проверки качества тестируемой программы.

Кроме того, тестирование чёрного ящика не сможет найти ошибки в модулях тестируемой программы, реализация которых основана на неправильных требованиях или ошибочной спецификации проекта. Если тестировщик критически относится к требованиям или спецификациям и руководствуется логическим анализом, то присутствует возможность обнаружить неправильные требования во время разработки тестов. В противном случае, подобные противоречия могут быть обнаружены уже только при эксплуатации готового продукта.

Наконец, тестирование чёрного ящика не может проверить дополнительную функциональность, находящуюся вне спецификаций. Иногда дополнительные функции не указаны или были упущены на этапе постановки требований, однако реализованы позднее при верификации или валидации. Такие функции могут быть покрыты тестовыми наборами только случайным образом, причем это никак не будет учтено при оценке качества соответствия требованиям.

Несмотря на указанные недостатки, метод черного ящика занимает центральное место в тестировании, так как он отслеживает основную цель разрабатываемого ПО – полное соответствие спецификации требований. Кроме того, данный подход может использоваться еще в процессе разработки программы, когда итоговый код программы не до конца реализован. Однако наилучший результат может быть получен, когда функциональный подход реализуется в сочетании со структурным подходом на основе метода белого ящика, описанным в следующем разделе.

Структурное тестирование

В основе тестирования структурным методом лежит взаимодействие со структурой тестового кода, откуда и происходит его название. Данный метод также называют белым ящиком, так как для него необходимо иметь доступ к исходному тестируемому коду, и, в некоторых случаях, должна присутствовать возможность манипулировать им.

Стоит учитывать, что ожидаемые результаты (конечная цель тестирования) должны определяться с использованием требований или спецификаций, а не самого кода. Это делается для того, чтобы решить, привело ли выполнение определенной части кода к ошибке или дефекту. Подход белого ящика может быть использован для оценки качества ПО определёнными критериями, например, критериями покрытия операторов или ветвей. Основная цель метода состоит в том, чтобы достичь заранее установленного минимального значения покрытия кода во время тестирования [52].

1.3 Генерация тестовых данных

Генерация тестовых данных – сложный и трудоемкий процесс, требующий больших усилий. Поэтому автоматизация этого процесса, хотя бы частичная, является актуальной исследовательской задачей, решение которой могло бы повысить эффективность тестирования программного обеспечения. Одной из целей автоматической генерации тестовых данных является создание такого множества тестовых наборов, которое обеспечило бы достаточный уровень качества конечного продукта путем проверки большей части различных путей кода, т.е. обеспечило бы максимальное покрытие кода в соответствие с выбранными критериями оптимальности (например, критерии покрытия операторов или ветвей). Подобрать такие наборы данных вручную является трудоёмкой задачей, поэтому в работе предлагается автоматизация данного процесса с использованием генетического алгоритма.

1.3.1 Граф потоков управления и критерии покрытия

Структурный подход к тестированию позволяет взаимодействовать с тестируемым кодом SUT при подборе наборов данных. Одним из способов визуализации кода является граф потоков управления (ГПУ, Control-flow graph, CFG), который определяется как направленный граф $CFG = (V, R, v0, vE)$, где V – набор узлов графа, R – подмножество декартова произведения $V \times V$, определяющее бинарное отношение на V (множество ребер графа), $v0$ и vE – входной и выходной узлы, соответственно, $v0 \in V,\ vE \in V$.

Ограничимся подмножеством структурных конструкций С-подобного языка программирования. Предположим, что тестируемый код может включать операторы присваивания (включая операторы выполнения функций), условные операторы (*if-then-else*), операторы циклов (*for* и *while*). Узел в V соответствует наименьшей исполняемой части оператора, т.е. соответствует множеству последовательно выполняемых операторов присваивания, операторам ввода или вывода, или части <выражения> от *if-then-else* или *while* операторов. Ребро (ветвь) графа $(v_i,\ v_j)$ соответствует возможной передаче управления от узла v_i к узлу v_j. Каждая ветвь в графе потоков управления может быть помечена предикатом, определяющим условия, при которых эта ветвь будет пройдена при очередном запуске программы.

Таким образом, можно определить путь P в графе, являющийся набором узлов $P = \langle v0,\ v_{i_1}, \ldots,\ v_{i_k}, \ldots, vE \rangle$, таких что $\left(v_{i_k},\ v_{i_{(k+1)}}\right) \in R$. Тестовый набор x_i инициирует прохождение по определённому пути P_i, то есть можно говорить, что тестовые наборы позволяют обеспечить покрытие определённых узлов графа, расположенных на данном пути. В книге для подготовки специалистов к сертификации в области тестирования [52] предлагаются критерии покрытия, которые хорошо себя проявляют в динамическом тестировании.

На вершине иерархии находится критерий покрытия всех внутренних условий, требующий обеспечения того, чтобы каждая комбинация значений для логических переменных в каждом условии выполнялась хотя бы один раз. Обеспечение этого критерия гарантирует покрытие по критериям, расположенным ниже в иерархии, однако его реализация предполагает генерацию множества тестовых наборов, являющихся избыточными. Внизу иерархии находится критерий покрытия функций (без раскрытия внутренней структуры функций), для которого требуется только, чтобы каждая функция вызывалась один раз в рамках набора тестов. Данный критерий не исследует покрытие внутренней структуры функций, поэтому является недостаточным.

Между этими крайними уровнями, находятся критерии покрытия путей, покрытия ветвей и покрытия операторов, в порядке следования иерархии. Покрытие путей определяет покрытие ветвей и покрытие операторов, т.е. стоит на более высоком уровне иерархии. Однако реализация этого критерия предполагает перебор всех возможных путей тестируемого кода, что представляется, с одной стороны, вычислительно сложной задачей, с другой – избыточной, так как многие пути отличаются малым количеством операторов.

Покрытие операторов основано на подсчете операторов SUT, которые будут выполняться, когда код запускается на конкретном тестовом наборе, по сравнению с общим количеством операторов. Различные пути кода определяются условиями и циклами. Следовательно, цель покрытия операторов – выполнить как можно больше операторов SUT по сравнению с общим количеством операторов:

$$\text{Покрытие операторов} = \frac{\text{Количество покрытых операторов}}{\text{Общее количество операторов}} * 100\%. \quad (1.1)$$

Этот подход покрытия также называется C0-покрытием и является относительно слабым. Более строгий критерий покрытия – это покрытие ветвей, также называемое C1-покрытием, которое определяется следующим образом:

$$\text{Покрытие ветвей} = \frac{\text{Количество покрытых ветвей}}{\text{Общее количество ветвей}} * 100\%. \qquad (1.2)$$

Значение покрытия ветвей всегда будет ниже покрытия операторов. Однако разница между этими критериями заключается больше в игнорировании выгруженных ветвей (*if* без *else*) в условных операторах, что не особенно важно. Поэтому в рамках данной работы для определения покрытия используется критерий покрытия операторов (1.1).

В качестве примера представлена иллюстративная программа для вычисления цены продукта в зависимости от товаров в корзине для интернет-магазина (Рисунок 1.4), заимствованная из книги для сертификации тестировщиков [52]:

```
int calculate_price(int baseprice, int specialprice, int extraprice, int extras, int
discount)
    {
        int addon_discount;
        int result;
        if (extras >= 3 && extras < 5) addon_discount = 10;
        else if (extras >= 5)
            if (baseprice >= 10000) addon_discount = 20;
            else addon_discount = 15;
        else addon_discount = 0;
        if (discount > addon_discount)
            for (int i = 0; i < discount - addon_discount; i++)
            {
                addon_discount += i;
                if (addon_discount >= 30)
                    addon_discount = 30;
            }
        result = Convert.ToInt32(baseprice / 100.0 * (100 - discount)
        + specialprice
        + extraprice / 100.0 * (100 - addon_discount));
        return (result);
    }
```

Рисунок 1.4 – Исходный код программы для вычисления цены

Программа позволяет определить итоговую цену товара в зависимости от исходной цены товара, дополнительных затрат и скидки. Размер скидки определяется количеством дополнительных товаров в заказе. Программа состоит из нескольких вложенных друг в друга условий, и одного цикла. Рисунок 1.5 показывает построенный для данной программы граф потоков управления

.

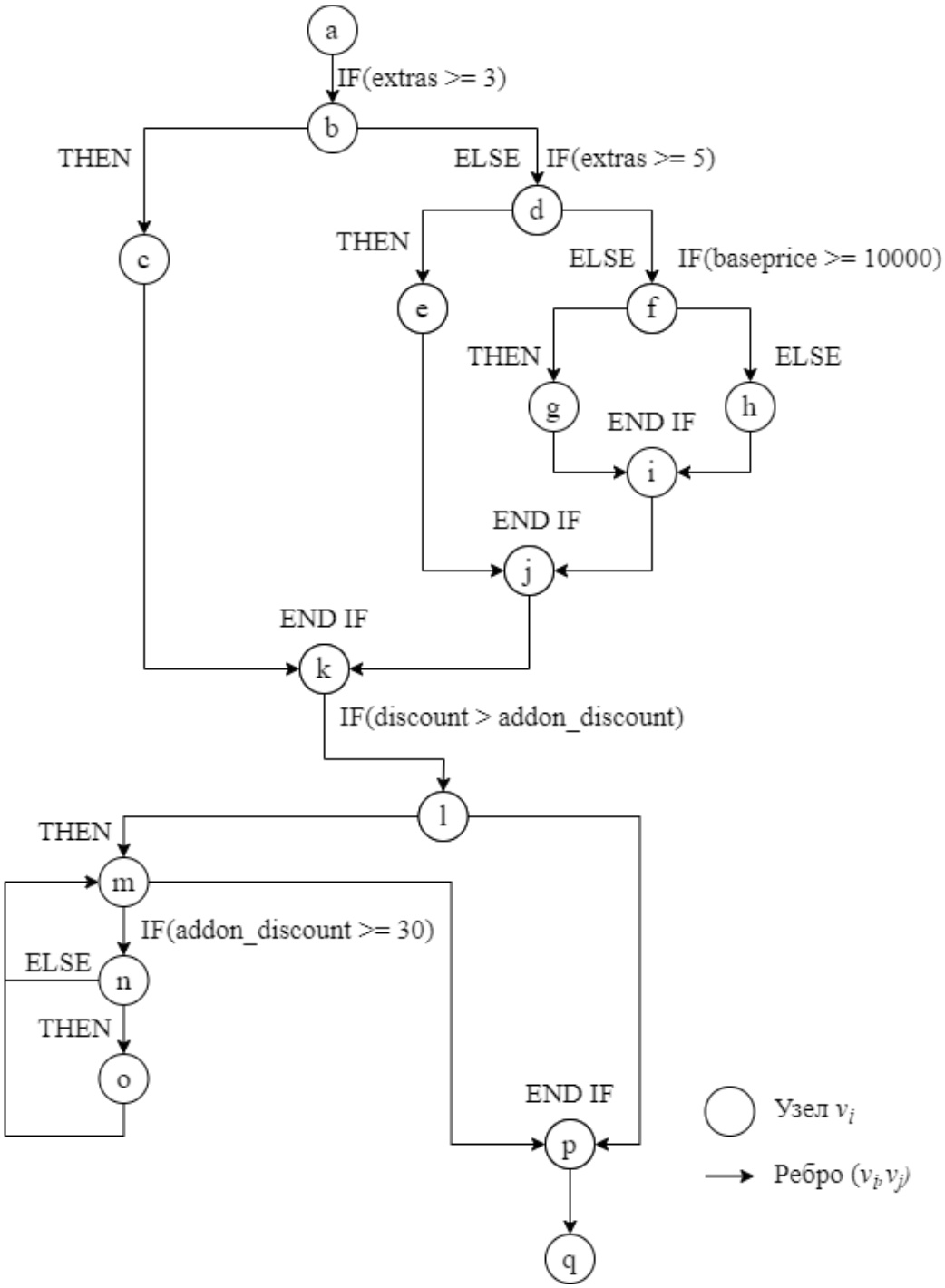


Рисунок 1.5 – Граф потоков управления для программы вычисления цены

Рисунок 1.6 отражает различные пути выполнения программы в зависимости от подобранных тестовых данных. Всего было использовано восемь тестовых наборов, которые проходят по четырем различным путям кода.

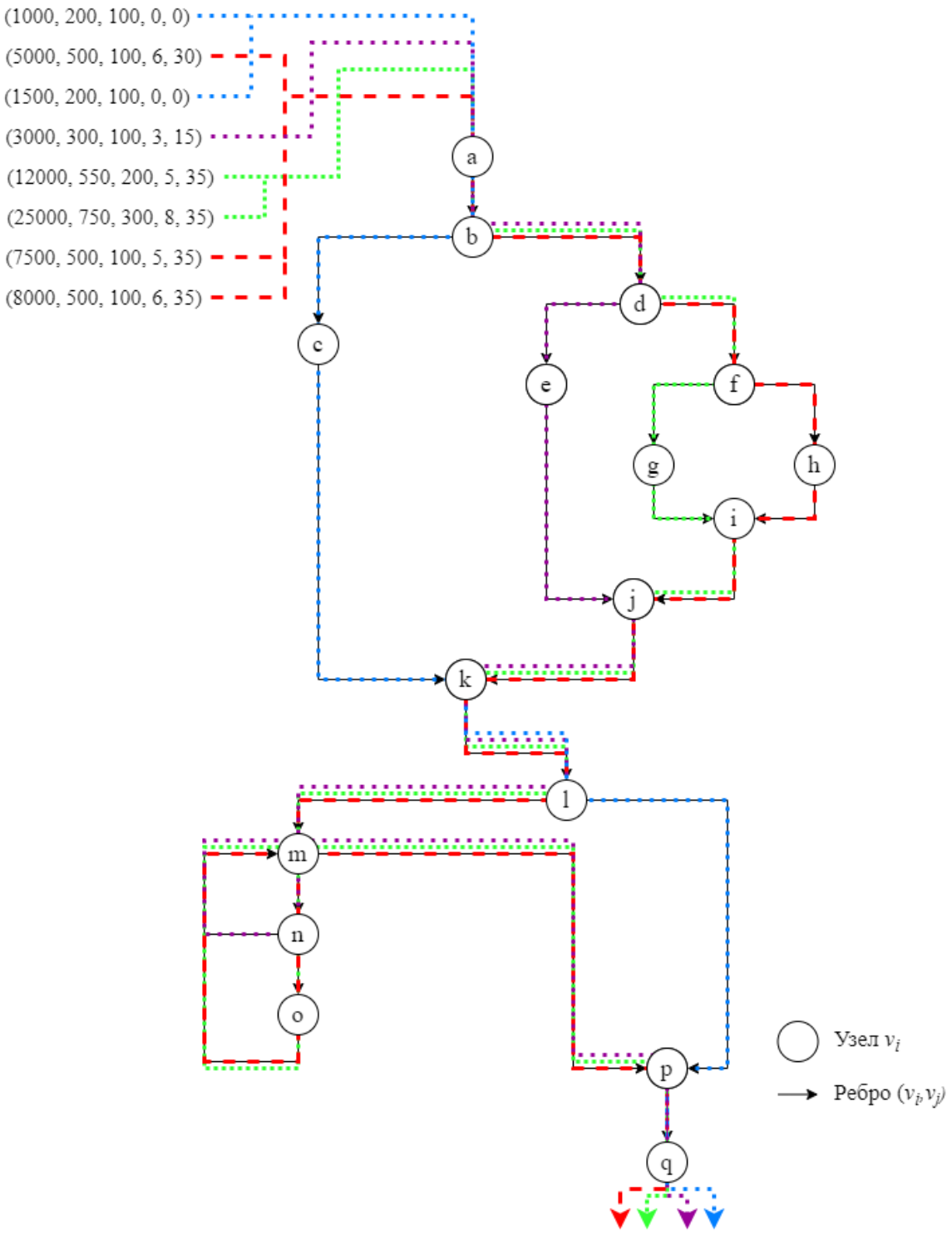


Рисунок 1.6 – Тестовые наборы и пути выполнения кода

В программе были сгенерированы данные для прохода по четырём путям. Проход по пути A обеспечивает покрытие подмножества операторов

$\{a, b, d, f, h, i, j, k, l, m, n, o, p, q\}$, проход по пути B – покрытие подмножества операторов $\{a, b, d, f, g, i, j, k, l, m, n, o, p, q\}$, проход по пути C – покрытие подмножества операторов $\{a, b, d, e, j, k, l, m, n, p, q\}$, последний путь D обеспечивает покрытие подмножества операторов $\{a, b, c, k, l, p, q\}$.

Таким образом, в данным случае достаточно четырех тестовых наборов, которые обеспечивают полное покрытие операторов при проходе по четырем путям, хотя потенциальное число путей – 12, то есть в 3 раза больше для этого простого графа. В случае большего размера кода мы получаем экспоненциальную зависимость числа путей от числа и глубины ветвлений в программе. Поэтому, хотя критерий покрытия путей может дать возможность выявления потенциально большего числа ошибок в программе (представляется, что эти дополнительные ошибки будут носить характер исключений), вычислительные затраты на их выявление будут чрезмерно высоки. Поэтому в данной работе используется критерий покрытия операторов как наиболее реалистичный и достаточно результативный (с точки зрения выявления ошибок) критерий для генерации тестовых наборов данных.

1.3.2 Обзор исследований в области генерации тестовых данных

В статье [53] были выделены три основных типа генераторов тестовых данных: генераторы на основе отслеживания путей кода, генераторы на основе спецификации требований и псевдослучайные генераторы. Генераторы на основе спецификации требований ставят критерием обеспечение наиболее полного удовлетворения требований к ПО в условиях функционального тестирования. Генераторы на основе отслеживания путей тестируемого кода, используемые в структурном тестировании (тестировании белого ящика), позволяют построить более совершенный набор тестов, удовлетворяющий критериям качества, использующим метрики сложности тестируемого кода. Использование псевдослучайных генераторов [54] – это простейшая слепая стратегия, которая

применяется в компаниях по разработке прикладного ПО. Однако есть несколько довольно успешных улучшений этого метода, основанных на использовании более совершенного статистического моделирования. В статье [55] предлагается усовершенствовать классический псевдослучайный метод с использованием оператора случайного блуждания, который оказался очень эффективным даже по сравнению с более сложными методами оптимизации. Другая стратегия развития псевдослучайного метода – использование современных статистических моделей. Например, в [56] специально определённая Марковская модель используется для проверки надежности беспилотных авиационных систем. Эксперименты показали, что предложенный метод позволяет снизить избыточность количества тестовых наборов при сохранении значения покрытия.

Совместное использование тестирования чёрного ящика и тестирования белого ящика иногда называется тестированием серого ящика. Тестирование серого ящика измеряет качество тестовых данных с использованием спецификации программного обеспечения, как в тестировании чёрного ящика, но также позволяет взаимодействовать со структурой тестируемой программы, как в тестировании белого ящика. В [57, 58] предлагается подход серого ящика для генерации тестовых наборов, исходя из спецификации программного обеспечения, полученной перед этапом программирования.

Одной из самых продвинутых моделей для определения спецификации требований, подходящей для тестирования серого ящика, – это диаграммы UML. Есть несколько исследований в области генерации тестовых данных, основанных на использовании UML. Например, в [59, 60] предлагается использовать генетические алгоритмы для генерации триггеров в диаграммах UML, позволяющие определить критический путь в тестируемой программе. В статье [61] предлагается улучшенный метод, также основанный на генетическом алгоритме, цель которого заключается в подборе множества тестовых наборов для параллельных путей. Помимо диаграмм UML, спецификации могут быть

отображены в виде дерева классификации [62]. В [63] была рассмотрен проблемы при построении деревьев для генерации тестовых данных, и предложен алгоритм интегрированного дерева классификации, а в [64] разработан прототип системы ADDICT (анг. AutomateD test Data generation using the Integrated Classification-Tree methodology).

Наличие тестовых наборов перед написанием кода помогает разработчикам контролировать процесс программирования в соответствии со спецификациями требований. Однако использование уже написанного кода в структурном тестировании позволяет получить более совершенный набор тестовых данных, одновременно обладающий как свойством неизбыточности наборов, так и оптимальным покрытием кода. Исторически первым подходом к генерации тестовых данных с помощью отслеживания путей кода был статический метод, основанный на использовании символьного выполнения, рассмотренный в статьях [65, 66, 67, 68, 69]. При символьном выполнении программа запускается с использованием символьных значений переменных вместо фактических, в рамках которого проводится формирование и проверка разрешимости в символьном виде системы ограничений. Входные переменные тестируемого кода (наборы тестовых данных) должны быть получены с учётом данных ограничений.

Существуют разные подходы для разрешения подобной системы ограничений. В [70] был реализован набор инструментов (вместе называемых Godzilla), который обеспечивает автоматическое формирование системы ограничений и нахождения её решений в качестве тестовых наборов. Тестовые наборы могут быть использованы при проведении юнит-тестирования и модульного тестирования. В [71] предложено использовать логическое программирование в ограничениях и символьное выполнение для решения проблемы. В [72] правила обработки ограничений используются для помощи при верификации проблемных частей тестируемого кода.

Несмотря на большое число исследований, посвященных статическому подходу к тестированию на основе символьного анализа, принципиальным моментом, существенно ограничивающим возможности его применения на практике, является проблема вычислительной сложности символьных вычислений даже для задач относительно небольшой размерности. Поэтому в настоящее время наиболее эффективным и применимым на практике является динамический подход, основанный на фактическом выполнении программы при некоторых значениях входных переменных и последующем анализе потока данных. Тестовые данные определяются фактическими значениями входных переменных. Одним из первых ориентированных на выполнение подходов был цепной метод [73], в котором используется анализ зависимостей данных для направления процесса поиска тестовых наборов. Анализ зависимостей позволяет автоматически определять последовательность операторов, которая должна быть инициирована до выполнения текущего оператора.

Исследования методов, ориентированных на выполнение с использованием диаграмм потоков данных, проводились в статьях [74, 75], где также исследуется применение гибридных подходов. Например, подход, предложенный в [76], объединяет случайную стратегию, динамическое символьное выполнение и стратегию на основе поиска. В [77] предлагается гибридный подход генерации тестовых данных, в основе которого находится меметический алгоритм. В [78] проведено сравнение различных методов генерации тестовых данных, включая генетические алгоритмы, случайный поиск и другие эвристические методы.

Что касается методов оптимизации, то в настоящее время эволюционные подходы зарекомендовали себя как мощный инструмент для генерации тестовых данных. Множество публикаций по этой теме посвящено исследованию применения генетического алгоритма (ГА) для решения проблемы генерации тестовых данных; например, в статьях [78, 79, 80, 81]. Однако стоит отметить, что чаще всего при использовании ГА исследователи ограничиваются методами

поиска тестовых наборов для одного пути, в которых нет возможности выбора среди множества путей, как в случае целенаправленной генерации [82]. В результате работы ГА из последнего поколения выбираются хромосомы, то есть наборы тестовых данных, наиболее подходящие для прохождения по определённому пути – это может быть как самый сложный путь, определённый метриками сложности кода, либо путь, установленный исследователем. Если цель состоит в том, чтобы сгенерировать множество тестовых наборов, обеспечивающих полное покрытие всех путей или, по крайней мере, некоторых из самых сложных путей программы, то классический ГА не подходит для этой задачи, поскольку в итоговой популяции тестовые наборы будут тяготеть в сторону значений, инициирующих прохождение по самому сложному пути кода. Другими словами, в процессе работы алгоритм будет формировать множество схожих тестовых наборов, которые не в состоянии обеспечить полное покрытие кода.

Указанную проблему смещения исследователи пытались решить использованием других эволюционных методов. В [83] предложен алгоритм имитации отталкивания, в основе которого лежит система частиц, для автоматической генерации ориентированных на разнообразие тестовых наборов (Diversity-Oriented Test Sets, DOTS), полученных путем итеративного увеличения разнообразия случайно взятых тестовых наборов.

В работе [84] сравниваются два вычислительных метода, ГА и метод роя частиц (Particle swarm optimization, PSO) для генерации тестовых данных. Было замечено, что PSO создает более точные тестовые примеры, чем ГА, но PSO может генерировать тестовые наборы только для дискретных переменных, в то время как ГА может рассматривать в качестве тестовых входных данных как дискретные, так и непрерывные переменные.

В [85] был предложен GPSMA (алгоритм смешанного генетического роя частиц) с использованием индивидуального режима обновления для замены операции мутации в ГА на основе разделения популяции. В [86] предлагается на

основе классического генетического алгоритма разделять популяцию на «семьи», оказывая влияние на эффективность сходимости путем скрещивания в семье, сохраняя разнообразие популяции смешиванием между семьями. В [87] был предложен подход интеграционного тестирования объектно-ориентированных программ на основе связывания с использованием PSO.

При определенном успехе гибридных эволюционных подходов к увеличению разнообразия популяции проблема автоматизации генерации тестовых наборов далека от окончательного решения. Следует отметить, что в большинстве работ используется простая формулировка функции приспособленности, которая выражает степень покрытия отдельного пути тестовым примером, хотя высказывается предположение, что возможно увеличение разнообразия популяции за счет специального вида функции приспособленности [88, 89], однако весомых подтверждающих результатов в этом направлении до сих пор не было получено.

Выводы по главе 1

В данной главе были приведены основные понятия в области программной инженерии как системной методологии разработки программного обеспечения, рассмотрены модели жизненного цикла и основные этапы разработки ПО. Выделен этап тестирования как один из наиболее затратных по времени и ресурсам и исследована проблема генерации наборов тестовых данных. Критерии качества тестовых наборов определены как критерии покрытия графа потока управления анализируемой программы (критерии покрытия кода), представлен иллюстративный пример графа и его покрытия множеством тестовых наборов.

Анализ существующих исследований, методов и подходов в области применения методов автоматической генерации тестовых данных показал, что в настоящее время повсеместно используемым в производстве программного обеспечения подходом является применение слепой стратегии псевдослучайной генерации тестовых данных, причем данная стратегия применяется в тестировании

как белого, так и черного ящиков (с использованием тестируемого кода или без использования). В то же время проводимые в настоящее время научные исследования показывают, что существуют подходы, разработка и применение которых может значительно улучшить качество генерируемых тестов, выражаемого в степени покрытия ими тестируемого кода. Особый интерес представляют подходы на основе анализа кода программ (тестирования белого ящика), так как извлеченная из текста программы информация может способствовать дальнейшему увеличению показателя качества покрытия кода.

Исторически первыми появились статические методы символьного анализа кода программы. Генерация тестовых данных в результате такого анализа сводилась к автоматическому формированию и разрешению в символьном виде системы уравнений и неравенств, получающихся логическим объединением и пересечением всех содержащихся в тестируемой программе условий, накладываемых на переменные.

Несомненным достоинством статического подхода является получение результатов в символьном виде, что дает аналитикам возможность определять так называемые домены – области значений переменных тестовых наборов, которые гарантируют проход вычислений по необходимым частям кода. Однако принципиальным моментом, существенно ограничивающим возможности применения символьного статического подхода на практике, является проблема вычислительной сложности символьных вычислений даже для задач относительно небольшой размерности. Поэтому в настоящее время более реалистичным и эффективным для практического использования в компаниях по производству ПО является динамический подход, основанный на фактическом выполнении программы при некоторых значениях входных переменных и последующем анализе потоков данных.

Наиболее перспективными методами реализации динамического подхода к генерации тестовых данных являются эволюционные методы оптимизации.

Эволюционная парадигма, которая лежит в основе генетического алгоритма, в качестве исходной точки отсчета использует множество случайных тестовых данных, полученных путем применения псевдослучайных генераторов, после чего проводится последовательная «эволюция» данных с целью улучшения качества покрытия тестируемого кода.

В связи с вышеизложенным возникает предположение о возможности адаптации генетического алгоритма для реализации идеи эволюционного улучшения тестовых данных с точки зрения максимизации покрытия ими тестируемого кода. Целью исследования, таким образом, является разработка и исследование методов автоматической генерации тестовых данных на основе модификаций генетического алгоритма для наиболее полного покрытия программного кода.

Для достижения сформулированной выше цели требуется решить ряд задач.

Так, необходимо исследовать возможности применения генетического алгоритма в области подбора входных тестовых наборов данных, определить специфику применяемых эволюционных операций, сформулировать выражение для функции принадлежности ГА на основе исследования и выбора соответствующих метрик оценки сложности путей программного кода. Далее, необходимо исследовать возможности модификации ГА для получения множества тестовых наборов, обеспечивающих наиболее полное покрытие программного кода. Здесь представляется целесообразным разработать и исследовать различные варианты функции приспособленности ГА для обеспечения наибольшего покрытия за счёт большего разнообразия множества сгенерированных тестовых наборов. Наконец, необходимо реализовать приложение для генерации множества наборов тестовых данных с использованием предложенных методов и алгоритмов, с использованием которого провести детальное исследование их эффективности для решения поставленных задач.

ГЛАВА 2 ГЕНЕТИЧЕСКИЙ АЛГОРИТМ В ЗАДАЧЕ ГЕНЕРАЦИИ ТЕСТОВЫХ ДАННЫХ

Первые исследования в области искусственного интеллекта и эволюционных вычислений пришлись на 50-е года прошлого века и ставили своей целью моделирование процессов эволюции [90, 91], в том числе создание искусственной жизни [92]. Особенную популярность эволюционным алгоритмам придала изданная в 1975 году книга «Адаптация в естественных и искусственных системах» [93] Джона Холланда (John Holland). Именно он ввёл понятие генетического алгоритма, как его принято понимать в настоящее время [94, 95]. Предыдущие попытки описать генетические алгоритмы использовали преимущественно только операцию мутации в качестве движущей силы эволюции. Холланд же сосредоточится на генетическом операторе скрещивания, который производит рекомбинацию генетической информации родителей для получения совершенно нового решения в виде потомков.

Впоследствии исследователи обнаружили, что генетические алгоритмы оказываются способом поиска решений для проблем, которые не могут быть решены другими методами, или могут быть решены за слишком длительный промежуток времени [96]. Генетические алгоритмы могут одновременно работать с различными решениями из пространства решений (параллелизм), оперировать с различными видами данных, как с дискретными, так и с непрерывными. Эти преимущества позволяют генетическим алгоритмам «создавать потрясающие результаты, когда традиционные методы оптимизации терпят неудачу» [97].

Одной из задач в области разработки программного обеспечения, в которой обосновано использование ГА – это генерация тестовых данных. Действительно, структура современных программ может быть логически сложной и разветвленной, и, следовательно, может содержать огромное число путей выполнения, инициированных входными значениями. В этом случае вместо традиционных

методов оптимизации целесообразно использование эволюционных подходов на основе эвристических решений.

В данной главе представлена формальная постановка задачи генерации тестовых данных для ее решения с помощью генетического алгоритма, исследованы особенности применения основных генетических операций. Проанализирована и реализована возможность определения вида функции приспособленности на основе метрик оценки сложности кода. Предложена версия ГА для решения задачи поиска тестового набора, покрывающего наиболее сложный путь, рассмотрен один из вариантов ее расширения для покрытия множества путей. Исследования, проведённые в данной главе, опубликованы в работах [7–12, 17–23].

2.1 Основные понятия генетического алгоритма

Терминология ГА заимствована из биологии и генетики [98]. Основные понятия ГА применительно к задаче генерации тестовых данных, представлены в таблице 2.1.

Таблица 2.1 – Основные понятия генетического алгоритма

Термин ГА	**Применительно к генерации тестовых данных**
Ген – одна компонента хромосомы	Входная переменная тестируемой программы
Хромосома – это упорядоченный набор генов, определяющих особь биологической популяции	Набор тестовых данных / Вектор входных переменных
Популяция – множество хромосом	Множество наборов тестовых данных

Классический генетический алгоритм предполагает преобразование генов в вид битовой строки. Джон Холланд определил, что представление значений генов в виде битовой строки позволяет существенно разнообразить популяцию [99]. Тем не менее, существуют и другие способы представления переменных, например,

использование кода Грея [100]. Для решения некоторых задач преобразование значений переменных в двоичный вид может быть существенно осложнено, поэтому гены могут быть представлены в виде исходных значений. Такой тип ГА называется непрерывным.

Для решения проблемы генерации наборов тестовых данных в работе было решено использовать именно непрерывный ГА, чтобы иметь возможность анализировать программы с различными типами данных на входе без необходимости преобразовывать их в битовую строку. Кроме того, при большом количестве входных переменных общая длина битовых строк может быть достаточно велика. Так как для к аждого из битов необходимо проводить эволюционные операции, длинные битовые строки могут негативно сказаться на общей производительности алгоритма.

Связь между различными компонентами непрерывного ГА представлена на рисунке 2.1. Разным оттенком выделяются различные значения входных переменных тестируемой программы, принимающие значения из непрерывных множеств.

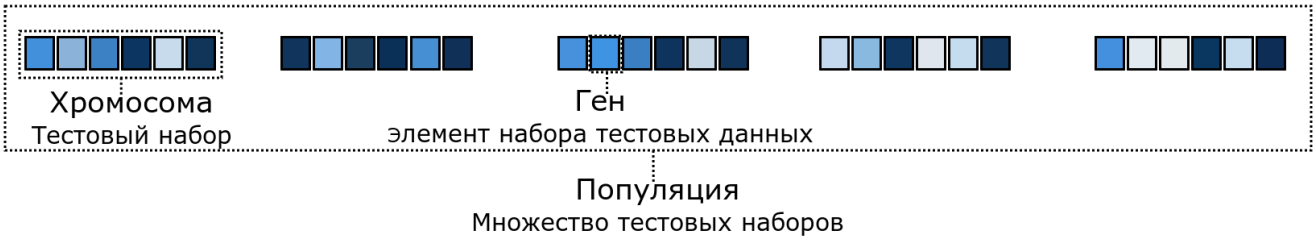


Рисунок 2.1 – Основные компоненты генетического алгоритма

Генетический алгоритм особенно предпочтителен при решении нестандартных задач, задач с неполными данными, или задач, для которых невозможно применение оптимизационных методов из-за сложности реализации или длительности выполнения [101, 102]. Де Йонг в своем исследовании [103] предположил, что размер популяции должен сохраняться в пределах 50-100

хромосом, чтобы оставаться «здоровой». Преимущество использования большого количества хромосом в одном поколении заключается в наличии множества точек в пространстве решений, что увеличивает скорость сходимости метода за счёт параллельного нахождения новых решений. Недостатком является существенное снижение производительности работы алгоритма (п. 3.2.1). Кроме того, подход, представленный в [103], несколько устарел, и часто количество хромосом определяется спецификой решаемой задачи. Так, в работах [104–108] показано, что количество хромосом может различаться в зависимости от конкретных условий.

Генетический алгоритм является нелинейным методом поиска решений, который может хорошо работать даже при наличии множества параметров и переменных. Основу его работы составляют эволюционные операции, которые применяются к популяции потенциальных решений итеративным способом, создавая новые поколения популяции при поиске оптимального решения. Тот факт, что множество точек в пространстве решений оценивается параллельно, позволяет определить генетический алгоритм как метод глобальной оптимизации.

Основной цикл генетического алгоритма представлен на рисунке 2.2. «Эволюция» решений основана на двух основных операциях – мутации и скрещивания. Несмотря на случайный характер этих операций, ГА не относят к классу методов случайного поиска, так как при формировании новых решений с лучшими параметрами используются ретроспективные знания, накопленные в родительской популяции. Таким образом, в каждом последующем поколении популяция претерпевает искусственную эволюцию. Относительно хорошие решения воспроизводятся, а относительно плохие вымирают и заменяются более приспособленными потомками.

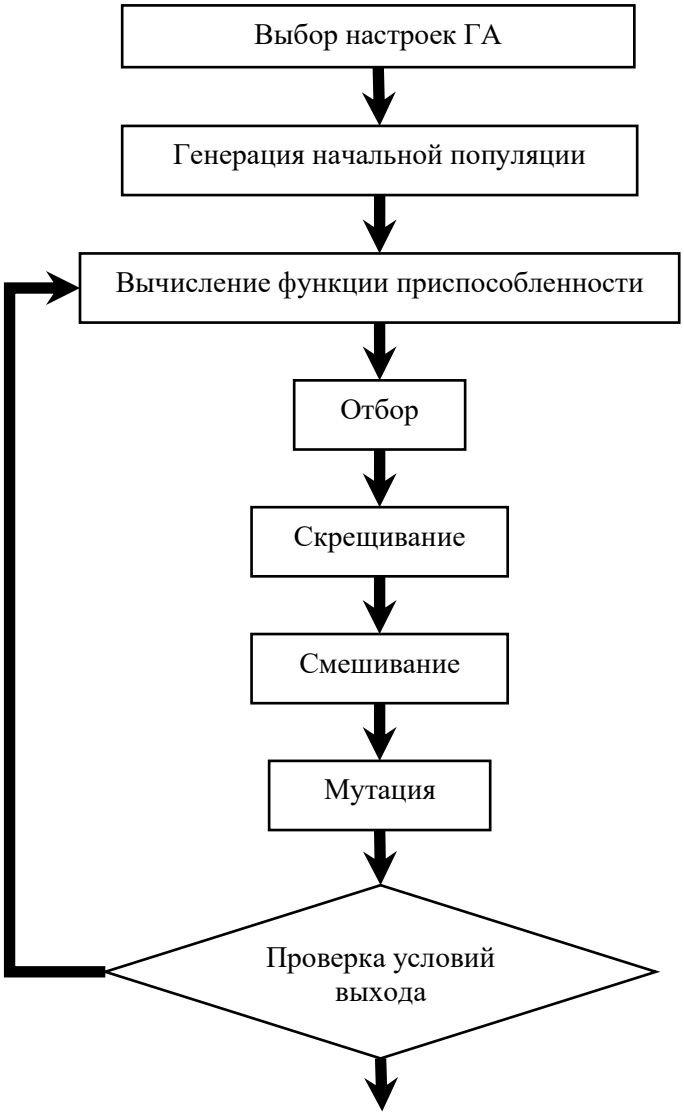


Рисунок 2.2 – Графическое представление работы ГА

2.2 Адаптация генетического алгоритма для генерации наборов тестовых данных

2.2.1 Формальная постановка задачи генерации тестовых данных

Как было определено в главе 1, в работе используется динамический подход к генерации данных, который основан на фактическом выполнении кода и динамическом анализе потока данных. В результате такого подхода определяется путь в графе потоков управления, по которому прошли вычисления,

инициированные определённым, рассматриваемым на конкретной итерации, набором тестовых данных.

Как было описано в п. 1.3.1, путь P определяется набором узлов $P = \langle v0,\ v_{i_1},\ldots,\ v_{i_k},\ldots, vE\rangle$, соединенными дугами графа. Путь P достижим, если существует входной тестовый набор, приводящий к прохождению потока управления по этому пути, в противном случае путь P недостижим.

Входные переменные тестируемого кода – это либо переменные var_j , $j = \overline{1,N}$, которые появляются в операторе ввода, например, $read(var_j)$, либо входные параметры процедур, инициирующие прохождение вычислений по пути P. Входные переменные могут быть разных типов, например, целые, действительные, логические и т.д. Определим $(var_1, var_2, \ldots, var_N)$ – вектор входных переменных для тестируемого кода. Область определения входных переменных $D = D_1 \times D_2 \times \ldots \times D_N$, где D_i – область определения входной переменной var_i.

Цель задачи автоматической генерации данных – найти множество тестовых наборов $\{x_1, x_2, \ldots, x_m\},\ x_i \in D$, которые инициируют прохождение по заданному множеству достижимых путей. Целевое множество может содержать один путь, несколько путей, или все множество достижимых путей (полное покрытие кода).

Качество хромосом (тестовых наборов) в популяции определяется с помощью функции приспособленности. На каждой итерации ГА значение функции приспособленности особей в популяции предположительно возрастает, что означает усовершенствование наборов тестовых данных с точки зрения выбранного критерия качества. Напомним, что в качестве критерия качества тестовых наборов данных был определен критерий покрытия ими операторов тестируемой программы (раздел 1.3.1).

В данной главе мы ставим локальную задачу определения набора тестовых данных, обеспечивающих выход на один путь программы. Это может быть либо конкретный заданный путь графа, либо, например, самый сложный (длинный) путь

программы, получающийся в результате выполнения множества вложенных условий, накладываемых на входные переменные. В этом случае в качестве функции приспособленности ГА удобно использовать функцию следующего вида:

$$\mathrm{F}(x) = \sum_{j=1}^{n(x)} w_j(x), \qquad (2.1)$$

где $w_j(x)$ – ненулевые веса операторов, расположенных на пути графа, инициированном тестовым набором $x \in D$, $n(x)$ – это количество операторов на этом пути.

Перепишем формулу (2.1) в виде, определяющем присутствие каждого оператора тестируемой программы. Это позволит учесть в явном виде вес каждого оператора, и в дальнейшем использовать эту формулу для решения задачи покрытия множества путей.

Итак, было определено, что выполнение тестируемой программы зависит от значений N входных переменных $var_1, var_2, \dots, var_N$. Тогда хромосома x_i из популяции $\{x_1, x_2, \dots, x_m\}$ представлена вектором размерности N:

$$x_i = \left[var_1^i, var_2^i, \dots, var_N^i\right].$$

где var_j^i – составляющие хромосому гены, $j = \overline{1, N}$.

В графе потоков управления операторы кода представлены в виде узлов, а потоки управления между операторами представлены в виде ребер (дуг направленного графа). В таком случае, один путь P_i графа представляется как непрерывное выполнение множества связанных друг с другом узлов, а разные входные тестовые наборы данных приводят к проходу по разным путям графа, обеспечивая выполнение только определенных (не всех) операторов тестируемой программы.

Введем обозначение $g(x_i)$ – вектор, являющийся индикатором покрытия операторов, инициированного определенным тестовым набором x_i:

$$g(x_i) = \big(g_1(x_i), g_2(x_i), \dots, g_n(x_i)\big),$$

где n – количество операторов тестируемой программы, а

$$g_j(x_i) = \begin{cases} 1, \text{если путь, инициированный набором } x_i, \text{ проходит через узел j} \\ 0, \quad \text{в противном случае.} \end{cases}$$

Присваивая веса различным операторам, можно учесть тот факт, что разные пути выполнения тестируемой программы имеют разную сложность. Больший вес назначается критическим операторам, которые являются частью путей, наиболее подверженных ошибкам, или выполняемых наиболее часто.

Введём понятие неразличимых хромосом, которое будем использовать в дальнейшем.

Определение. Хромосомы x_{i_1} и x_{i_2} называются неразличимыми, если $g(x_{i_1}) = g(x_{i_2})$, т.е. неразличимые хромосомы инициируют прохождение вычислений по одному и тому же пути на графе потоков управления.

Если обозначить вектор весов операторов тестируемой программы через $(w_1, w_2, \dots, w_n)$, то функция приспособленности (2.1) для отдельной хромосомы x_i может быть записана следующим образом

$$\mathrm{F}(x_i) = \sum_{j=1}^{n} w_j g_j(x_i), \qquad (2.2)$$

где w_j – вес j-го оператора, g_j – значение индикатора покрытия, n – количество операторов.

Чем больше сумма весов операторов, выполняемых на пути, инициированном тестовым набором x_i, тем большее значение имеет функция приспособленности $\mathrm{F}(x_i)$. Рассмотрим, как веса w_j могут быть получены при помощи использования различных метрик оценки сложности кода.

2.2.2 Метрики оценки сложности кода

Метрики оценки сложности кода, используемые для определения качества разработанного программного обеспечения, могут быть использованы для определения весов операторов w_j в формуле (2.2). Для решения задачи генерации

тестовых данных наибольший интерес представляют количественные метрики, так как для их вычисления необходим только тестовый код. Можно выделить следующие количественные метрики, которые могут быть использованы при вычислении весов.

Метрика SLOC

Количество строк кода (Source Lines Of Code, SLOC) является наиболее простой в понимании метрикой, которая учитывает только факт выполнения оператора, а не количество выполнений. Это позволяет оценить, сколько именно операторов покрывает тестовый набор, и, таким образом, определить самый длинный путь. На рисунке 2.3 показана схема простой программы, цифрами показан рассчитываемый метрикой SLOC вес.

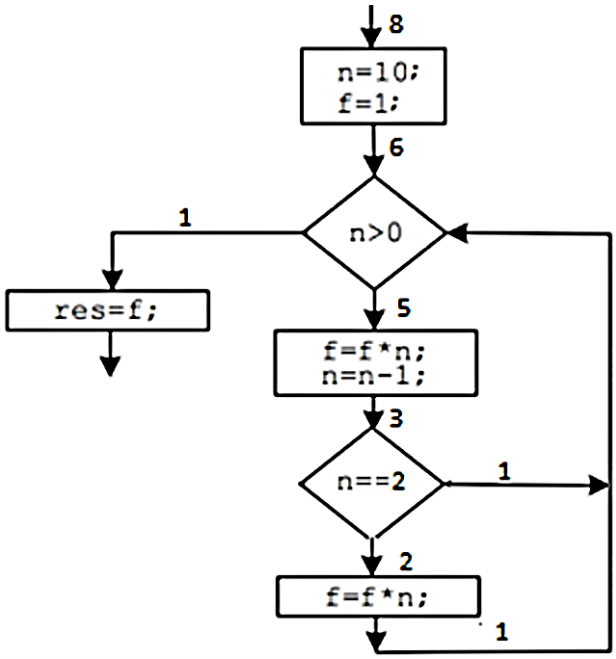


Рисунок 2.3 – Пример назначения весов по метрике SLOC

ABC метрика

Для метрики ABC или метрики Фитзпатрика (анг. Fitzpatrick) определяется три различных показателя: количество присваиваний переменным значений (показатель A); количество логических операторов (показатель B); количество

выполнений функций (показатель C). Таким образом, эта метрика описывается вектором с тремя значениями, например, ABC = (3,4,7). Пример распределения показателей показан на рисунке 2.4.

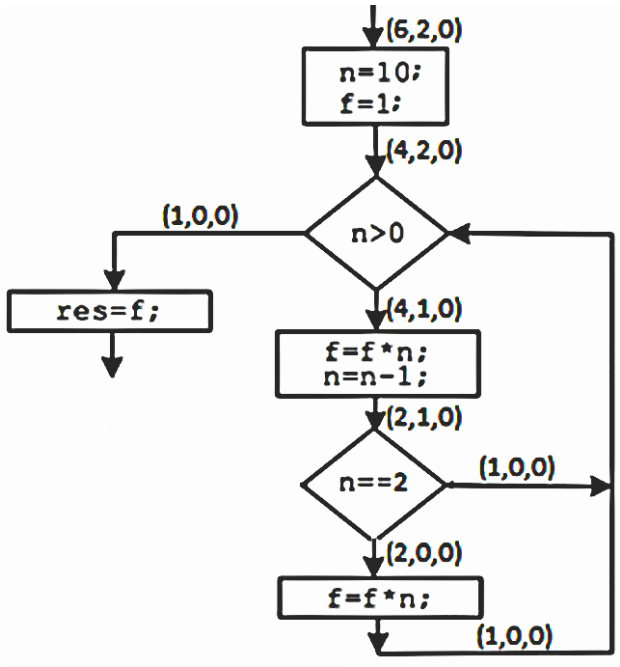


Рисунок 2.4 – Пример назначения весов по метрике ABC

В качестве показателя сложности программного кода, высчитывается только одно значение – квадратный корень из суммы квадратов всех трех значений. Рассчитать функцию приспособленности по метрике ABC можно по следующей формуле:

$$F = \sqrt{A^2 + B^2 + C^2}.$$

Метрика Джилба (Jilb).

Используется для определения сложности программного кода на основе количества условных и цикличных операторов. В данном случае F – относительная сложность, рассчитываемая по формуле:

$$F = {C_l}/{n_l},$$

где n_l – общее число покрытых операторов программы, C_l – абсолютная сложность исходного кода по Джилбу, определяемая как количество покрытых условий и

циклов в тестируемом коде. То есть для вычисления данной метрики важным фактором является покрытие условий и циклов (Рисунок 2.5).

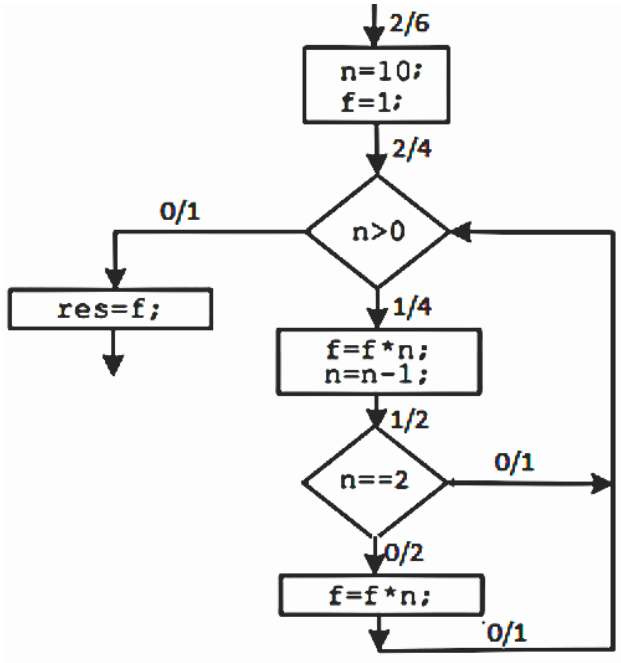


Рисунок 2.5 – Пример назначения весов по метрике Джилба

Несмотря на простоту вычисления, метрика позволяет отразить сложность понимания кода и трудоёмкость его написания. Связано это с тем, что большое число условий и циклов в коде увеличивает число возможных путей выполнения программы, что существенно затрудняет возможности предсказать, каким образом данные будут обрабатываться, и какая часть кода в результате будет выполнена после прохождения нескольких условий.

Метрика NOD

Модифицированная метрика для вычисления весов NOD (Nested Operation Division, разделение вложенных операторов) предложена для алгоритма генерации тестовых данных и может настраиваться модификаторами снижения веса [80].

В основе реализации метрики NOD лежит смещение акцента при генерации тестовых данных на операторы более высокого порядка, т.е. операторы, находящиеся внутри как можно меньшего числа условий или циклов.

Соответственно, под вложенными операторами понимаются те операторы, которые находятся внутри определённых циклов или условий. Если оператор находится внутри множества условий, его срабатывание ограничено множеством проверок, поэтому он выполняется реже и менее приоритетен в процессе генерации данных.

Вышеописанная логика используется в классическом варианте метрики NOD для определения весов операторов. Однако, в зависимости от требований к получаемым данным, наибольший интерес для тестирования могут представлять, наоборот, глубоко вложенные операторы. Тем не менее, в данной работе мы будем использовать классический вариант метрики.

Алгоритм назначения весов с использованием метрики NOD:

– Назначается вес первого оператора. Изначальный вес определяется количеством операторов в коде. Для программ небольшого размера достаточно веса в 10 единиц. Для программ большого размера можно использовать вес в 100 единиц. В данной работе в качестве начального используется значение 100;

– Каждому последующему оператору назначается вес предыдущего, то есть назначение весов производится последовательно. Если нет условий или циклов, вес передаётся без изменений;

– Изменение веса определяется срабатыванием циклов или условий. В случае, если вложенные операторы находятся внутри условия с одной ветвью выполнения или внутри цикла, то их вес снижается до 80% от предыдущего оператора. Для условий с двумя и более ветвями вес равномерно распределяется по условиям – для двух ветвей 50%/50%, для трёх 33%/33%/33% и т.д.

– Каждый уровень вложенности учитывается при определении веса, например, для оператор вложенного в два условия с одной ветвью вес будет равен 100 * 80% * 80% = 64.

Таким образом обеспечивается снижение весов и концентрация алгоритма на операторы более высокого порядка, так как подобные операторы должны выполняться в первую очередь.

Анализ метрик для задачи генерации тестовых данных

При использовании метрик NOD, SLOC и ABC получаемые тестовые наборы не слишком сильно отличаются друг от друга, что достаточно предсказуемо, так как в их реализации учитываются одни и те же показатели. Для данных метрик более свойственна концентрация на определённых частях кода. Использование метрики Джилба, в свою очередь, показало худшие результаты, так как в ней используются отличные от других метрик параметры оценки кода, которые не слишком подходят для алгоритма генерации данных.

Метрики ABC и SLOC достаточно схожи в реализации, так как обе учитывают только факт выполнения определённых операторов, а не их количество. Однако, метрика SLOC позволяет определить сложность кода лишь на основе его длины и, в целом, проще для понимания. Метрика NOD позволяет учесть поток работы программы и оценить тестовые наборы с учётом количества выполнений отдельных операторов. Поэтому в работе предлагается ограничиться двумя метриками при проведении исследований, а именно, метриками SLOC и NOD.

2.2.3 Основные этапы генетического алгоритма для генерации тестовых данных

Генетический алгоритм работает итерационно, выполняя на каждой итерации ряд последовательных этапов, пока не будут достигнуты условия завершения работы. С каждой новой итерацией ГА генерирует новое поколение на основе предыдущего (родительского). Основной цикл ГА для генерации тестовых данных включает следующие этапы, которые, за исключением первого этапа, выполняются итерационно до достижения заданного значения покрытия или числа поколений:

1. Инициализация. Исходная популяция формируется случайным образом с учетом ограничений на значения входных переменных. Объем популяции m выбирается на основе размера тестируемой программы (а именно, минимального количества различных возможных путей, по которым могут пойти вычисления).

2. Оценка популяции. Каждая хромосома популяции оценивается функцией приспособленности (например, функцией (2.1) в случае необходимости покрытия заданного пути P).

3. Селекция (Отбор). Лучшие 20% хромосом отбираются в неизменном виде для следующего поколения; остальные 80% хромосом следующего поколения будут получены в результате скрещивания. Данная пропорция получена эмпирически и позволяет обеспечить достаточное разнообразие популяции с высокой скоростью сходимости.

4. Скрещивание. Половина особей следующего поколения формируется путем случайного скрещивания 20% лучших хромосом предыдущего поколения друг с другом. Остальные хромосомы будут получены путем случайного скрещивания всех хромосом предыдущего поколения друг с другом. Скрещивание происходит путем выбора случайной константы $\beta_i \in [0,1]$ для каждого $l = \overline{1,N}$ и последовательного смешивания, где l-й ген потомка является линейной комбинацией соответствующих генов родительских хромосом:

$$var_l^{offspring} = \beta_l * var_l^{mother} + (1 - \beta_l) * var_l^{father}, l = \overline{1,N}.$$

5. Мутация. С заданной вероятностью мутации (0.05) каждый ген может изменить свое значение на случайное в рамках заданных ограничений. Основная цель мутации – достижение большего разнообразия.

6. Формирование тестовых наборов данных в виде пула элитных хромосом. В каждом поколении происходит отбор особей популяции в пул элитных хромосом. В пул попадают лишь те хромосомы, которые обеспечивают дополнительное покрытие кода по сравнению с предшествующим покрытием.

После того, как все этапы ГА были исполнены, определяется, выполняются ли условия завершения работы, или процесс переходит к следующей итерации. Итерационность ГА является тем фактором, который позволяет получать новые решения. Каждое новое поколение формируется на основе предыдущего, то есть

тестовые наборы предыдущего поколения участвуют в формировании новых наборов, обеспечивая таким образом «эволюцию» ранее полученных решений.

Для решения задачи генерации наборов тестовых данных главным условием завершения должно быть достижение полного покрытия исходного кода. Если в текущем поколении было получено множество тестовых наборов, обеспечивающих полное покрытие тестируемого кода, то работу ГА можно остановить.

Тем не менее, достижение полного покрытия кода не должно являться единственным условием прекращения работы алгоритма. Если в коде присутствуют недостижимые части, или части, выполняющиеся только при определённых условиях, не зависящих от процесса работы программы, то алгоритм может работать бесконечно долго, так и не достигнув полного покрытия. В таком случае, работа алгоритма должна ограничиваться количеством поколений.

Ограничение работы алгоритма количеством поколений является во многих случаях вынужденной необходимостью. Ввиду того, что начальная популяция задается случайным образом, а пространство решений может быть очень большим, для поиска тестовых наборов, обеспечивающих желаемый уровень покрытия, может понадобиться очень большое число итераций (поколений). Выход из алгоритма при достижении определённого числа поколений может не достигнуть цели полного покрытия, но позволяет получить приемлемый его уровень или, в случае необходимости, перенастроить параметры ГА для повторного запуска.

2.3 Особенности применения основных эволюционных операций в задаче генерации тестовых данных

Первым этапом работы ГА, как было сказано выше, является формирование начальной популяции. Стандартно в ГА начальная популяция определяется случайным образом, то есть первая популяция наполняется хромосомами с равномерно распределенными случайными значениями тестовых переменных в

рамках заранее заданных областей определения. Несмотря на то, что существуют и другие методы инициализации, именно случайный метод формирования начальной популяции используется в данной работе, так как он позволяет обеспечить достаточное для начальной популяции разнообразие получаемых тестовых вариантов. После того, как была сформирована начальная популяция, начинается последовательное выполнение основных эволюционных операций в соответствии с алгоритмом ГА. Далее рассмотрим особенности их применения в задаче генерации тестовых данных, использованные в настоящей работе.

2.3.1 Отбор (Селекция)

Хромосомы в популяции сортируются в соответствии со значением функции приспособленности. Сортировка позволяет разделить хромосомы на лучшие (с большим значением функции) и худшие (с меньшим значением функции), что является необходимым критерием для выполнения отбора.

Перед этапом отбора часть лучших хромосом переносится в новое поколение без изменений, тем самым формируя пул лучших хромосом. Остальные хромосомы нового поколения будут получены в результате скрещивания. Именно для определения пула подходящих для скрещивания хромосом производится операция отбора. Пул лучших хромосом (P_{best}) и пул хромосом для скрещивания (P_{cross}) могут как совпадать, так и не совпадать.

Главная цель отбора состоит в выборе пар родительских хромосом, которые будут использованы для формирования потомков. Можно выделить следующие основные методы отбора родительских пар [98]:

1. Последовательные пары. Для скрещивания отбираются последовательно расположенные в популяции хромосомы. Если популяция состоит из m хромосом $\{x_1, x_2, \dots, x_m\}$, то для хромосомы x_i отбирается следующая за ней x_{i+1}. Таким образом, один из родителей всегда будет выбран из подмножества нечётных хромосом $\{x_1, x_3, x_5, \dots\}$, а другой – из подмножества чётных $\{x_2, x_4, x_6, \dots\}$. Такой

способ отбора прост в понимании и реализации, но негативно сказывается на разнообразии популяции.

2. Случайное определение пар. Каждая хромосома из пула для скрещивания P_{cross} имеет одинаковую вероятность быть отобранной. Случайное определение позволяет обеспечить достаточно высокое разнообразие популяции, но может негативно сказаться на скорости сходимости при большом размере P_{cross} [109].

3. Метод рулетки (колеса рулетки). Данный метод функционально схож с предыдущим, но вероятность хромосомы быть отобранной задается пропорционально значению функции приспособленности. Применительно к задаче генерации тестовых данных, хромосомы, инициирующие прохождение по более сложным путям, имеют большую вероятность быть отобранными для скрещивания. Вероятность отбора может быть определена следующим образом:

a. Взвешивание по рангу. Ввиду того, что в популяции хромосомы отсортированы по значению функции приспособленности, позиция может выполнять роль ранга хромосомы. Вероятность отбора зависит от ранга и может быть описана, например, формулой (2.3).

$$P(x_i) = \frac{m_{cross} - m_i + 1}{\sum_{j=1}^{m_{cross}} m_j}, \tag{2.3}$$

где $P(x_i)$ – вероятность отбора тестового набора x_i, m_{cross} – количество хромосом в пуле для скрещивания, m_i – ранг хромосомы.

b. Взвешивание по значениям. В данном методе вероятность отбора (2.4) напрямую зависит от значения функции приспособленности

$$P(x_i) = \frac{F(x_i)}{\sum_{j=1}^{m_{cross}} F(x_j)}, \tag{2.4}$$

где $F(x_i)$ и $F(x_j)$ – значения функции приспособленности для x_i и x_j.

Данный метод лучше всего работает, когда популяция состоит из хромосом с несущественно различающейся функцией приспособленности. В

противном случае вероятность отбора хромосом с высоким значением функции приспособленности может быть существенно выше остальных.

4. Турнирный отбор. Особенностью турнирного отбора является предварительный выбор небольшого подмножества хромосом, в котором отбирается хромосома с наибольшей функцией приспособленности. Турнирный отбор является одним из наиболее успешных методов отбора, но для него необходимо определение дополнительного параметра размера подмножества предварительного выбора.

Методы отбора имеют свои преимущества и недостатки. Наибольший интерес представляют методы турнирного отбора и метод случайного определения пар. Первый метод позволяет разделить популяцию на отдельные подгруппы, в которых происходит отбор родителей, а второй – достичь большего разнообразия, что в задаче по генерации тестовых данных играет важную роль. Поэтому в работе предлагается гибридный метод отбора на основе данных методов.

Прежде чем приступать непосредственно к отбору, необходимо определить, какая часть нового поколения будет получена переносом лучших хромосом, а какая – в результате скрещивания. Эмпирически для задачи генерации тестовых данных были определены следующие параметры. В качестве лучших хромосом для переноса в новое поколение без изменений выбираются 20% хромосом с наибольшей функцией приспособленности, составляющих пул лучших P_{best}. Остальные 80% хромосом нового поколения будут отобраны с использованием гибридного метода:

– 50% хромосом для скрещивания выбираются только из пула P_{best};

– 50% хромосом будут отбираться из популяции в целом, то есть значение функции приспособленности не учитывается.

Скрещивание среди всех хромосом необходимо для достижения большего разнообразия популяции. Таким способом мы можем сохранить хромосомы, которые потенциально (на следующих итерациях) могут привести к

инициированию более сложных путей кода, выход на которые затруднен системой ограничений, содержащихся в условных операторах тестируемого кода. Таким образом, гибридный метод позволяет лучше исследовать тестовый код и отбирать хромосомы с целью достижения большего разнообразия.

После отбора пары родительских хромосом, они формируют одного потомка с использованием эволюционной операции скрещивания.

2.3.2 Скрещивание

Скрещивание позволяет получать новые тестовые наборы перемешиванием значений уже существующих. В терминах генетического алгоритма, две родительских хромосомы обмениваются генами для получения потомка. Основная цель использования скрещивания заключается в итерационном улучшении поколения последовательным скрещиванием лучших хромосом для получения лучших решений. В данной работе скрещивание используется также для увеличения разнообразия популяции гибридным методом отбора.

Существуют различные методы скрещивания. При их описании, для удобства, определим одну из родительских хромосом как материнскую (x^{mother}), а вторую – как отеческую (x^{father}).

Скрещивание по одной позиции

Для получения потомков у родительских хромосом выбирается одна позиция, по которой происходит обмен генами. Стандартно используется одна статичная позиция для скрещивания, более продвинутым методом является определение позиции случайным образом.

Есть две родительские хромосомы, содержащих множество тестовых значений, – $x^{mother} = [val_1^{mother}, \dots, val_N^{mother}]$ и $x^{father} = [val_1^{father}, \dots, val_N^{father}]$ с количеством входных переменных N. При скрещивании данных хромосом по одной позиции r можно получить следующих потомков –

$x_1^{child} = [val_1^{mother}, \dots, val_r^{mother}, val_{r+1}^{father}, \dots, val_N^{father}]$ и $x_2^{child} = [val_1^{father}, \dots, val_r^{father}, val_{r+1}^{mother}, \dots, val_N^{mother}]$.

Графически скрещивание двух хромосом с 6 входными переменными и позицией скрещивания $r = 4$ показано на рисунке 2.6.

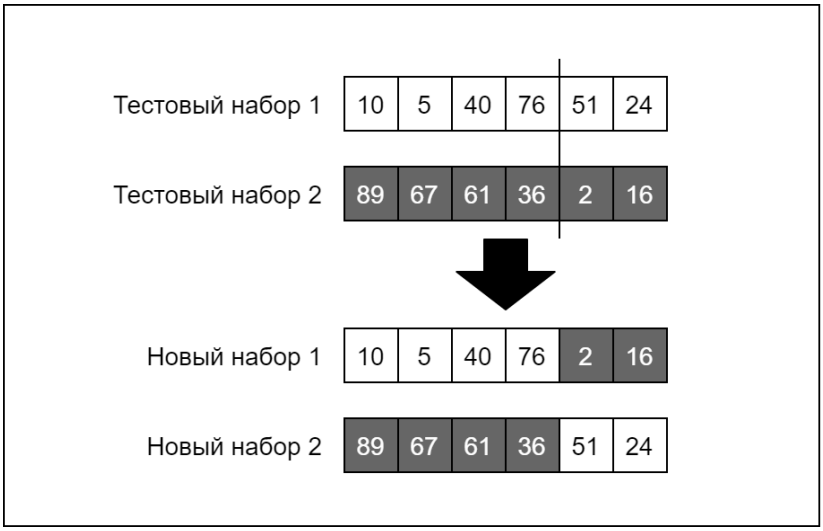


Рисунок 2.6 – Пример одноточечного скрещивания тестовых наборов

Применяя скрещивание по одной позиции, можно получить две новые хромосомы, но они не слишком сильно отличаются от родительских. В случае, если одна из частей хромосомы оказывает гораздо большее влияние на результат, то новые тестовые наборы, полученные таким образом, не будут вносить никакого разнообразия в популяцию, что негативно сказывается на работе алгоритма в целом. Одноточечное скрещивание является простым для понимания и реализации, но имеет слишком существенные недостатки.

Скрещивание по двум позициям

Являясь логичным развитием скрещивания по одной позиции, данный метод скрещивания позволяет достичь большего разнообразия за счёт обмена внутренними частями хромосом. Как и в предыдущем примере, предположим, что есть две родительских хромосомы – $x^{mother} = [val_1^{mother}, \dots, val_N^{mother}]$ и $x^{father} = [val_1^{father}, \dots, val_N^{father}]$. Первая позиция r для скрещивания может

принимать случайное значение в интервале ($1 \leq r < N-2$), вторая позиция l выбирается в интервале ($r+1 \leq l \leq N-1$). В результате могут быть получены следующие потомки:

$$x_{c1} = [val_1^{mother}, \dots, val_r^{mother}, val_{r+1}^{father}, \dots, val_l^{father}, val_{l+1}^{mother}, \dots, val_N^{mother}];$$

$$x_{c2} = [val_1^{father}, \dots, val_r^{father}, val_{r+1}^{mother}, \dots, val_l^{mother}, val_{l+1}^{father}, \dots, val_N^{father}].$$

Пример двухточечного скрещивания тех же хромосом, что и ранее, по двум позициям $r = 2$ и $l = 5$, представлен на рисунке 2.7. Здесь хорошо заметна разница меду одноточечным и двухточечным скрещиванием. При двухточечном скрещивании новые хромосомы будут существенно отличаться как друг от друга, так и от родительских особей.

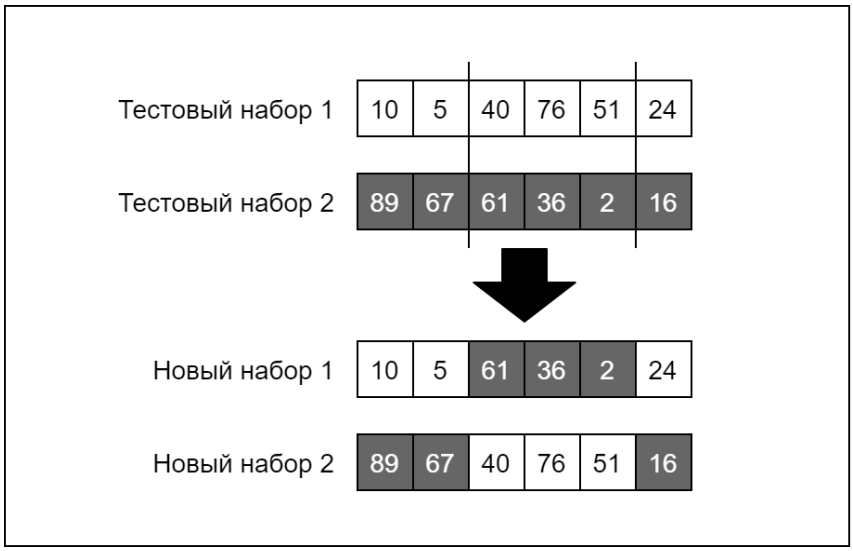


Рисунок 2.7 – Пример двухточечного скрещивания тестовых наборов

Унифицированное скрещивание

Очевидно, что если двухточечное скрещивание позволяет получать более разнообразные результаты, то при увеличении количества позиций для скрещивания возможно получать более отличающиеся тестовые наборы. Унифицированное скрещивание является типом скрещивания, при котором количество генов и количество позиций для скрещивания совпадает. Унифицированное скрещивание существенно отличается в подходе к получению

потомков – вместо разделения родительских хромосом на части, каждый ген случайным образом наследуется от одного из родителей. Ввиду того, что родители в ГА равнозначны, то каждый ген потомка с вероятностью $P(x^{mother}) = P(x^{father}) = 0.5$ возьмёт своё значение либо от одного, либо от другого родителя. Ввиду того, что в данном методе фактор случайности оказывает большее влияние, новые тестовые наборы могут существенно отличаться от родительских, что оказывает положительный эффект на разнообразие всей популяции (Рисунок 2.8).

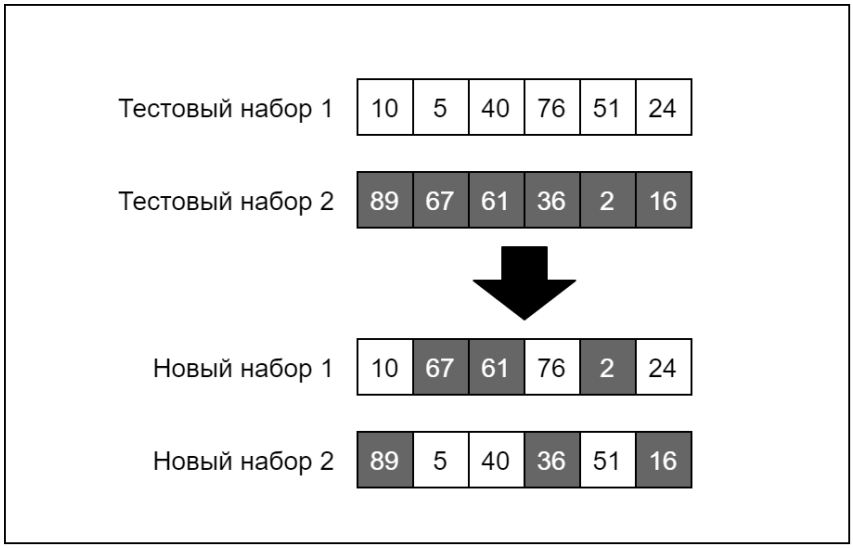


Рисунок 2.8 – Пример унифицированного скрещивания тестовых наборов

Данный метод является наиболее полным типом скрещивания, обеспечивающим наибольшее разнообразие потомков, поэтому именно он используется в качестве эволюционной операции скрещивания для генерации тестовых данных в данной работе.

Тем не менее, даже с использованием унифицированного скрещивания, возникает проблема недостаточного разнообразия популяции, в большой степени связанная с использованием непрерывного варианта ГА. Связано это с тем, что родительские хромосомы при скрещивании лишь обмениваются значениями, а это не позволяет получать новые значения генов в процессе скрещивания. В таком случае единственным механизмом, который вносит в популяцию новые значения,

становится мутация (п. 2.3.4), чрезмерное увеличение вероятности которой нежелательно из-за усиления при этом случайного характера эволюции. Чтобы нивелировать недостатки непрерывного ГА, предлагается использовать эволюционную операцию смешивания, которая позволяет вводить в популяцию новые значения в зависимости от значений генов родителей.

2.3.3 Смешивание

Как было сказано ранее, смешивание необходимо для получения новых значений генов в популяции, что существенно для генерации новых значений в непрерывном варианте ГА. В бинарном варианте ГА такой проблемы не возникает, что показано далее.

Предположим, что имеется два гена – значения какой-либо входной переменной тестируемого кода, 23 и 13 (Рисунок 2.9). При унифицированном скрещивании по 2 и 4 позициям в виде битовых строк можно получить два новых значения – 29 и 5. Таким образом, даже простое скрещивание двух хромосом из одного гена при бинарном ГА позволяет получать различные значения потомков.

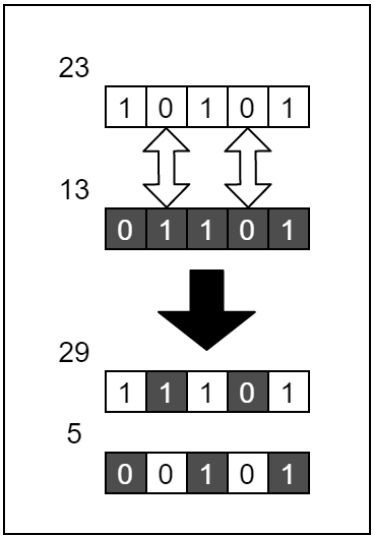


Рисунок 2.9 – Скрещивание двух чисел 23 и 13 в виде битовых строк

При скрещивании в непрерывном ГА происходит простой обмен генов, то есть не достигается дополнительного разнообразия. Фактически, потомки состоят

из такого же набора генов, что и родители. В этом случае велик шанс того, что потомки и родители могут оказаться неразличимыми (инициировать один и тот же путь на графе потоков управления).

Смешивание является механизмом, позволяющим получить отличное от родителей значение генов в процессе скрещивания. Можно выделить следующие методы смешивания:

– Прямое смешивание значений. Новое значение гена будет выбрано случайным образом в промежутке между родительскими значениями.

– Средние значения. Значение нового гена будет получено усреднением значений родительских генов. Очевидно, что такой метод плохо подходит для увеличения разнообразия, но может использоваться, например, для поиска подходящих границ входных переменных.

– Сдвигающееся смешивание. Значение нового гена формируется путем умножения случайной величины $\beta \in [0,1]$, на разницу между значениями родительских генов. Предполагая, что значение var_l^{mother} больше, чем var_l^{father}, данный вариант смешивания можно отобразить в виде

$$var_l^{offspring} = \beta_l * var_l^{mother} + (1 - \beta_l) * var_l^{father}, \; l = \overline{1,N}. \qquad (2.5)$$

Последний вариант (2.5) был выбран в диссертации в качестве основного метода смешивания.

2.3.4 Мутация

Мутация является эволюционной операцией, которая позволяет вносить в популяцию новые значения с помощью случайного изменения генов. Мутация применяется только для потомков, поэтому лучшие тестовые наборы, переносимые в новое поколение напрямую (без операции скрещивания), не будут изменяться.

Очень важно задать верную вероятность мутации, так как слишком маленький шанс может нивелировать преимущества мутации в целом, и алгоритм может не подобрать тестовые наборы для выхода на более сложные пути. Слишком

высокий шанс, наоборот, будет непредсказуемым образом изменять потомков, из-за чего нивелируются преимущества использования ГА по сравнению со случайным поиском.

Мутация применяется для каждого гена отдельно. В зависимости от реализации случайного шанса мутации, она может либо быть не применена к хромосоме вовсе, либо изменить только один ген (Рисунок 2.10), либо изменить несколько генов (Рисунок 2.11).

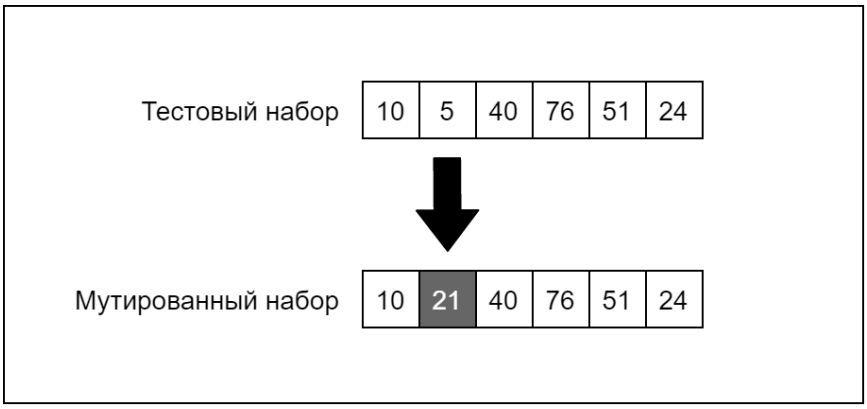


Рисунок 2.10 – Пример одноточечной мутации тестового набора, в котором значение второй переменной изменилось

На рисунке 2.11 из шести генов три поменяли свое значение, что, формально, уже может считаться другой хромосомой. Такое существенное изменение может негативно сказаться на сходимости ГА, поэтому не стоит задавать слишком высокую вероятность мутации.

Хоть мутация является достаточно мощным инструментом для обеспечения разнообразия популяции, им необходимо пользоваться осторожно. Мутация может как помочь в нахождении более сложных путей, так и осложнить поиск. Эмпирически, для задачи генерации тестовых данных, была определена оптимальная вероятность мутации в 5% (0.05), которая, в совокупности со смешиванием, позволила обеспечить достаточное разнообразие популяции.

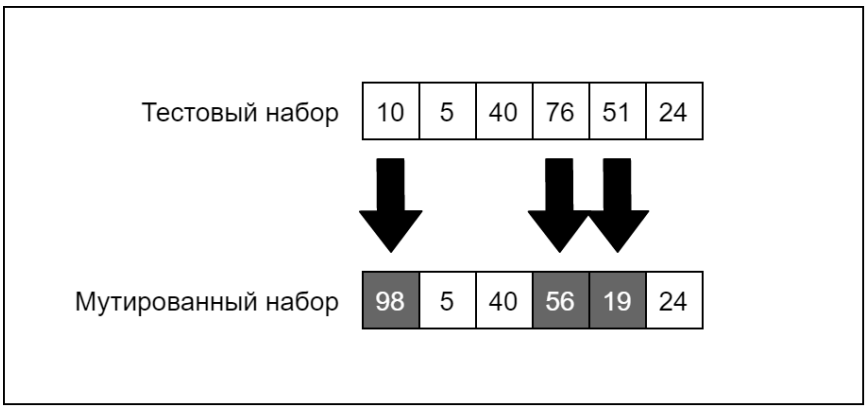


Рисунок 2.11 – Пример многоточечной мутации тестового набора, в котором значение нескольких переменных изменились

Таким образом, итерационно выполняя ранее рассмотренные эволюционные операции, алгоритм обеспечивает поиск тестовых данных.

2.4 Иллюстративные примеры использования генетического алгоритма для генерации тестовых данных

Для иллюстрации процесса генерации тестовых данных с помощью ГА приведём два иллюстративных примера. В первом примере используется формулировка функции приспособленности, отличная от используемой в диссертации, но позволяющая графически отобразить процесс поиска решений на двумерной плоскости. Второй пример позволяет понять процесс генерации данных для простого тестового кода, для которого нет необходимости использовать большой размер популяции.

2.4.1 Графическая иллюстрация получения тестовых наборов

Для наглядного представления алгоритма работы ГА можно использовать простую задачу на двумерном пространстве решений (Рисунок 2.12), т.е. с двумя входными переменными ($N = 2$) в тестируемом коде. На рисунке 2.12а представлена плоскость решений и сгенерированные случайным образом 10

тестовых наборов. Особи распределены равномерно по плоскости решения, но нет ни одной, которая бы была непосредственно в области решения.

После того, как хромосомы сгенерированы случайным образом, их необходимо оценить функцией приспособленности. В данном примере границы значений для выхода на самый сложный путь выделены большим чёрным кругом. Для наглядности будем использовать функцию приспособленности как расстояние до центра области решения вида (вместо использования функции (2.2), используемой в диссертации):

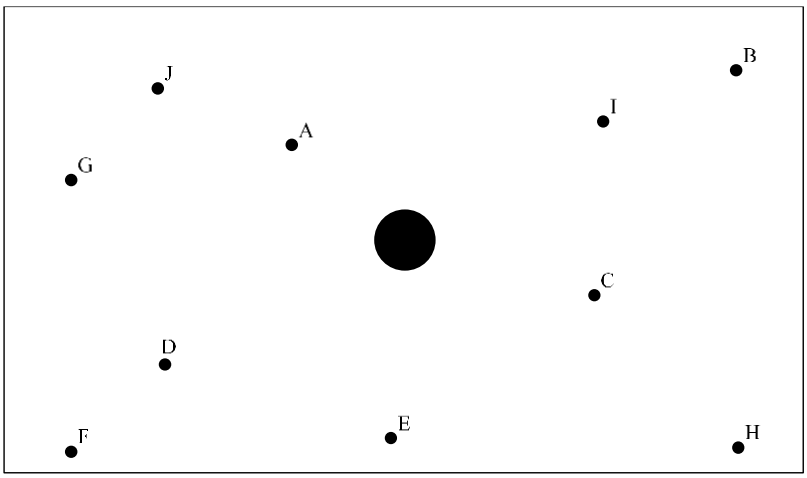
$$F(var1_i,\ var2_i) = \sqrt{(var1_i - var1_0)^2 + (var2_i - var2_0)^2}, \qquad (2.6)$$

где $(var1_i, var2_i)$ – значения i-ого тестового набора ($i = \overline{1,10}$), $(var1_0, var2_0)$ – позиция центра области решения.

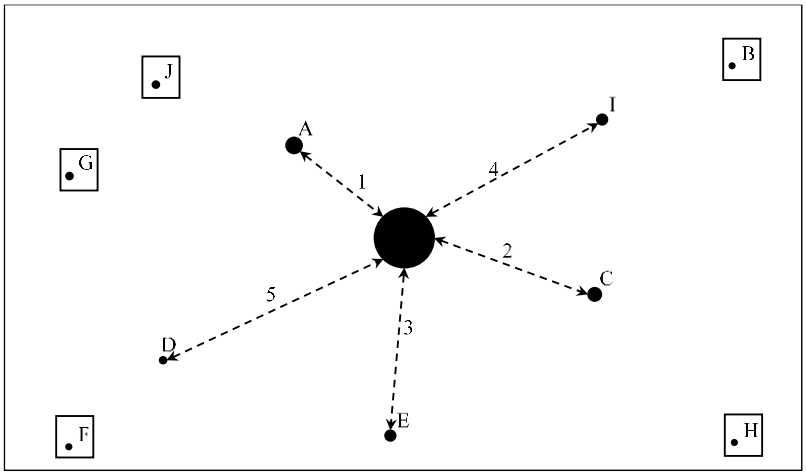
Таким образом, используя в качестве значений позицию точки на плоскости, можно визуализировать смещение позиций тестовых наборов в сторону решения. Так как происходит оценка расстояния, то при таком виде функции приспособленности решается задача минимизации. Оптимальное решение будет достигнуто для некоторого i^* при $F(var1_{i^*},\ var2_{i^*}) = 0$.

После того, как все особи были оценены функцией приспособленности, происходит сортировка решений от лучшего к худшему (Рисунок 2.12б). Худшие тестовые варианты здесь помечены квадратом, и они исключаются из популяции. Лучшие соединены пунктирной линией с центральным кругом и для каждой особи определён ранг в зависимости от расстояния до центра области решения.

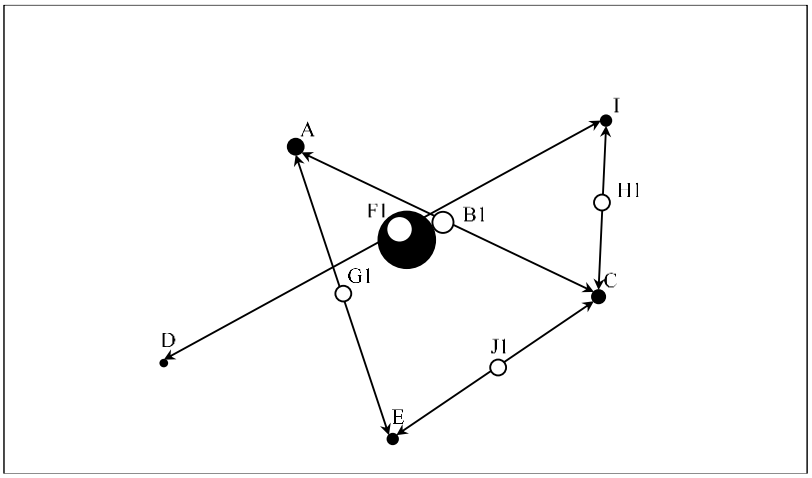
Отбор особей для скрещивания происходит случайным образом среди всех оставшихся особей. В результате для скрещивания выбраны пары (A, C); (A, E); (C, I); (C, E); (D, E) (Рисунок 2.12б). На рисунке они соединены линиями. Скрещивание проводится с применением смешивания значений родительских хромосом (метод среднего значения). Новые полученные особи отображены в виде белых кругов с чёрной обводкой.



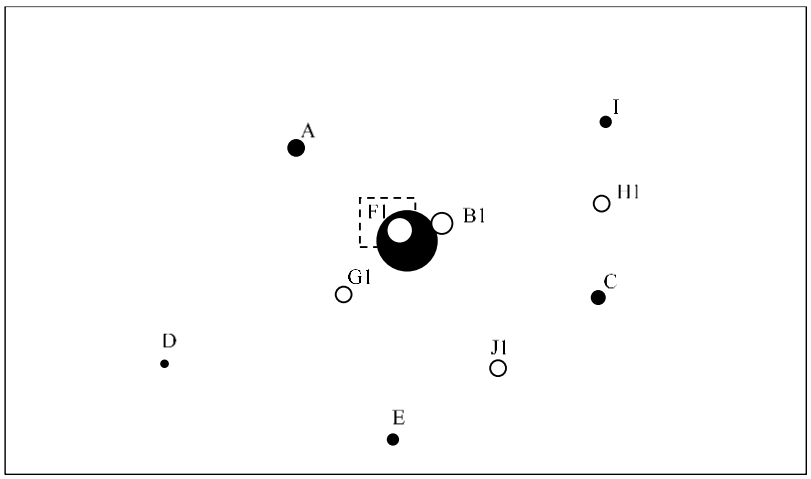
(а) Большой круг в центре – границы для выхода на самый сложный путь. Маленькие круги с латинскими буквами – сгенерированные особи



(б) Отбор лучших хромосом и исключение худших. Худшие выделены квадратами.



(в) Новые полученные хромосомы (белые круги с чёрной обводкой), являющиеся средним значением родительских.



(г) Результирующее поколение (F1 – решение задачи).

Рисунок 2.12 – Иллюстративный пример на двумерном пространстве решений

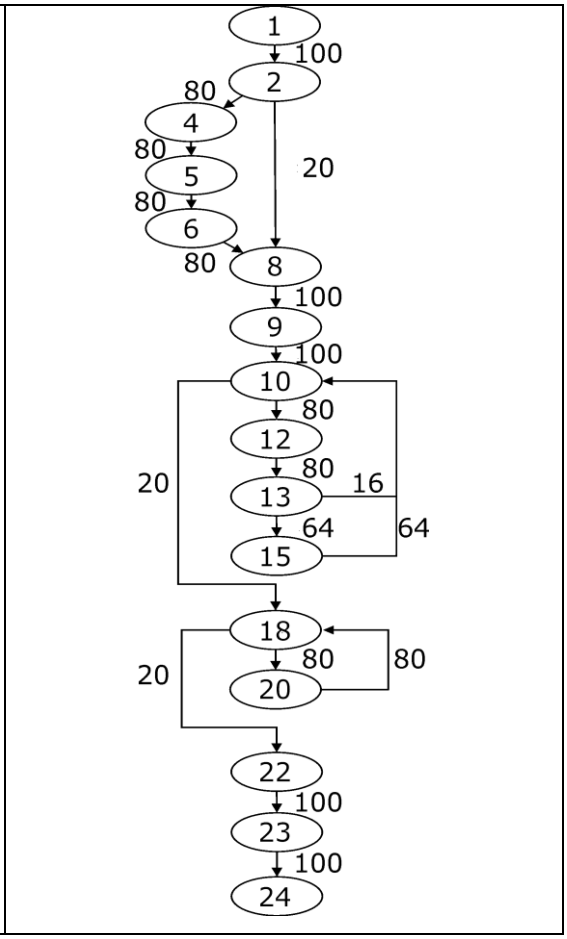
В результате скрещивания, были получены особи B1, F1, G1, H1, J1. Как было сказано выше, значения этих хромосом усреднены по сравнению с родительскими, поэтому они оказались посередине линий на пространстве решений. В итоге центр одной из новых особей, F1, оказался внутри области решения, поэтому мы можем считать её решением задачи (Рисунок 2.12г). Довольно интересным фактом является то, что хоть A и C являются лучшими случайно сгенерированными особями, лучшим решением в итоге оказалась особь F1, полученная скрещиванием худших особей. Это говорит о том, что оставлять только малую часть лучших хромосом может быть плохой идей, так как их скрещивание не гарантирует получение оптимального решения. В то же время, оставлять слишком много особей нецелесообразно, так как это может негативно сказаться на скорости сходимости.

2.4.2 Пример работы генетического алгоритма для генерации тестовых данных

На рисунке 2.13 представлен тестируемый код, написанный на языке C# (Рисунок 2.13а), а также построенный для него граф потоков управления с весами (Рисунок 2.13б), определенными на основе метрики оценки сложности кода NOD с начальным значением 100. В результате случайной генерации начальной

популяции было получено 4 набора тестовых данных (хромосом): (10,5,12); (3,4,10); (25,30,11); (5,3,17).

```
0.  public static void Main(int m, int i, int n) {
1.  int sum = 0, avg = 0;
2.  if (m <i)
3.  {
4.  use_m = m;
5.  }
6.  int[] a = new int[m];
7.  Console.WriteLine("Enter the Array Elements ");
8.  for (j = 0; j< m; j++)
9.  {
10. a[j] = i;
11. if (a[j] > n)
12. {
13. a[j] = n
14. }
15. }
16. for (k = 0; k < m; k++)
17. {
18. sum += a[k];
19. }
20. avg = sum / m;
21. Console.WriteLine("Average is {0}", avg);
22. Console.ReadLine();
23. }
```



(а) Тестируемый код (б) Граф с назначенными весами

Рисунок 2.13 – Тестируемый код

Для каждой хромосомы рассчитывается значение функции приспособленности, после чего они сортируются в порядке уменьшения ее значений. В таблице 2.2 показаны тестовые наборы, значение функции приспособленности и ранг. Лучшие хромосомы выделены курсивом и будут использоваться для скрещивания. В результате отбора для скрещивания были выбраны наборы 2 и 3. Два других набора исключаются, а популяция будет дополнена потомками отобранных хромосом. Для простоты иллюстрации здесь не используется механизм смешивания.

Таблица 2.2 – Исходные тестовые наборы данных (популяция)

№	**Набор данных X**	**F(X)**	**Ранг**
3	*(25,30,11)*	*1308*	*1*
2	*(3,4,10)*	*1196*	*2*
1	(10,5,12)	896	3
4	(5,3,17)	896	3

Скрещивание производится унифицированным методом, то есть хромосомы с тремя переменными могут быть разделены по первой и второй позициям. Обозначим первую родительскую хромосому x^{mother}, вторую – x^{father}. Родительские хромосомы переносятся в новое поколение без изменений. Ввиду того, что был задан небольшой размер популяции, добавление в популяцию только двух потомков не позволит наглядно отобразить процесс формирования нового поколения. Поэтому примем в качестве исключения, что новое поколение будет содержать 6 хромосом – 2 родительские и все 4 вариации потомков (обычно размер популяции не меняется от поколения к поколению).

Мутация каждого гена происходит с вероятностью 0.05 на интервале (0, 50). Очевидно, что при таком шансе мутации с малым размером популяции и количеством переменных, он является недостаточным для обеспечения существенного разнообразия. Все новые потомки показаны в таблице 2.3. Из-за небольшой вероятности мутации только один из тестовых наборов поменял своё значение (подчёркнут), остальные потомки не мутировали.

Таблица 2.3 – Новые наборы данных, полученные в результате скрещивания

№	x^{mother}	x^{father}	**Новый набор**	**Мутация**
1	(3,4,10)	(25,30,11)	(3,4,11)	(3,4,<u>23</u>)
2	(3,4,10)	(25,30,11)	(3,30,11)	(3,30,11)
3	(3,4,10)	(25,30,11)	(25,4,10)	(25,4,10)
4	(3,4,10)	(25,30,11)	(25,30,10)	(25,30,10)

Далее потомки оцениваются функцией приспособленности, и для них определяется ранг (Таблица 2.4).

Таблица 2.4 – Родительские хромосомы и их потомки

№	**Набор данных X**	**F(X)**	**Ранг**	**Поколение**
1	*(25,30,11)*	*1308*	*1*	*0*
2	*(3,30,11)*	*1308*	*1*	*1*
3	*(25,30,10)*	*1308*	*1*	*1*
4	(3,4,23)	1196	2	1
5	(3,4,10)	1196	2	0
6	(25,4,10)	896	3	1

В итоге, во втором поколении содержится уже 3 набора данных, которые проходят по самому сложному пути кода – (25,30,11), (3,30,11) и (25,30,10). Первый набор является лучшим из предыдущего поколения, однако два других варианта, (3,30,11) и (25,30,10), были получены в результате скрещивания.

На данном примере можно хорошо увидеть недостатки использования непрерывного ГА без механизма смешивания. На популяции маленькой размерности буквально за 1 поколение значения тестовых наборов начали повторяться. Вторая переменная, при последующем скрещивании, не будет изменяться, а третье будет колебаться, принимая всего два возможных значения (10, 11). Из-за небольшого шанса мутации алгоритм, скорее всего, будет формировать только неразличимые хромосомы. Однако для данного примера это и неважно, так как значение 1308 является оптимальным значением функции приспособленности для рассматриваемого примера, и оно и было получено сразу на этапе случайной генерации.

2.5 Исследование возможности применения алгоритма покрытия одного пути для решения задачи нахождения полного покрытия

В данном разделе исследуется возможность применения ранее разработанного алгоритма для нахождения множества тестовых наборов для полного покрытия тестируемого кода. Для этого будем использовать ранее предложенный вид функции приспособленности, который позволяет получить один тестовый набор, инициирующий прохождение по самому сложному пути.

Для начала опишем тестовые коды SUT1 и SUT2, которые будут использованы нами для проведения всех дальнейших исследований в диссертационной работе.

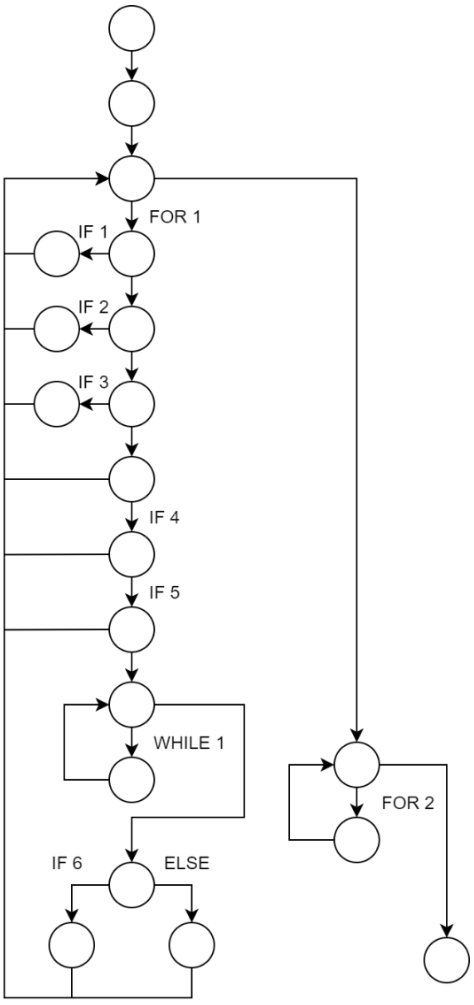
2.5.1 Описание исследуемых тестовых программ SUT1 и SUT2

Было разработано два тестовых кода, содержащих множество условных операторов, определяющих достаточное количество различных путей программного кода. Меньший по размеру исходный код SUT1 имеет две вариации, первая вариация – с возможностью получения 100% покрытия, вторая вариация – при наличии недостижимых частей кода (то есть достижение 100% покрытия невозможно). Исходные коды тестовых программ представлены в приложении А. На рисунке 2.14 показан граф потоков управления для первой вариации SUT1.

На вход программы подаётся 3 целочисленных переменных val1, val2 и val3. FOR 1 является основным циклом программы и содержит большинство операторов и условий, поэтому многие операторы будут выполняться несколько раз. Условия IF 1, IF 2 и IF 3 проверяются последовательно и требуют различные значения тестовых данных для их последовательного инициирования. Условие IF 6 будет достигнуто только в том случае, если оба IF 4 и IF 5 истинны и цикл WHILE 1 завершен.

Все условия в тестовом коде зависят только от входных переменных val1, val2 и val3, за исключением условия IF 3, в котором дополнительно проверяется

локальная переменная weight_count. Поэтому, если условие было выполнено хоть раз, то оно будет выполняться на всём протяжении работы цикла и в функции приспособленности будут учитываться только операторы внутри одного этого условия. Если ни одно из условий не было выполнено, то внутри цикла не будет выполнено ни одного оператора.

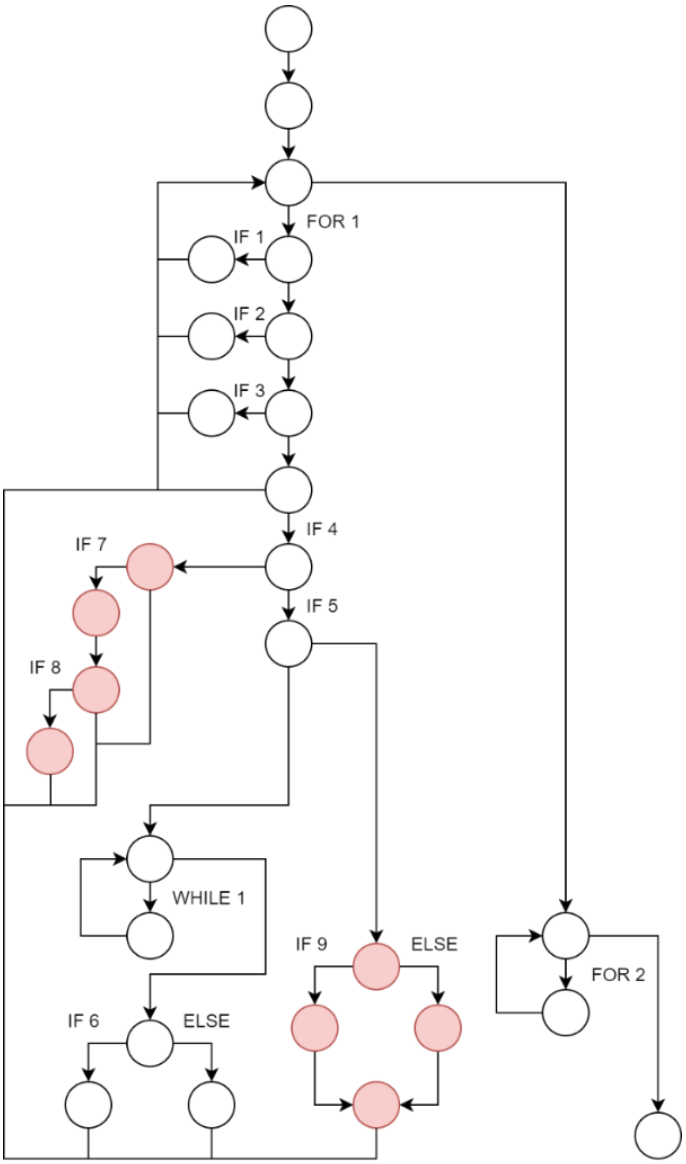


Input variables: val1, val2, val3
FOR 1: for (i = 0; i < 100; i++)
IF 1: (val1 > 5 & val1 < 60) | (val2 > 90 | val2 = 10)
IF 2: val1 = 60
IF 3: val3 > 5 & val1 < 30 & weight_count > 1
IF 4: val3 > 50
IF 5: val1 < val3
WHILE 1: while (n < 10)
IF 6: val1 > 50
FOR 2: for (i = 0; i < output.Count; i++)

Рисунок 2.14 – Граф и реализация условий и циклов для первой вариации SUT1

Вторая вариация SUT1 (Рисунок 2.15) полностью повторяет первую, за исключением того, что в нее были дополнительно введены недостижимые части кода, то есть операторы находятся внутри невыполнимых условий. В предложенной вариации условие IF 7, и, соответственно, условие IF 8, никогда не могут быть выполнены из-за проверок внутренней переменной *weight_count*,

которая изменятся в IF 4. Условие IF 9 недостижимо, из-за ограничений, вводимых на val2 и val3 предыдущими условиями.

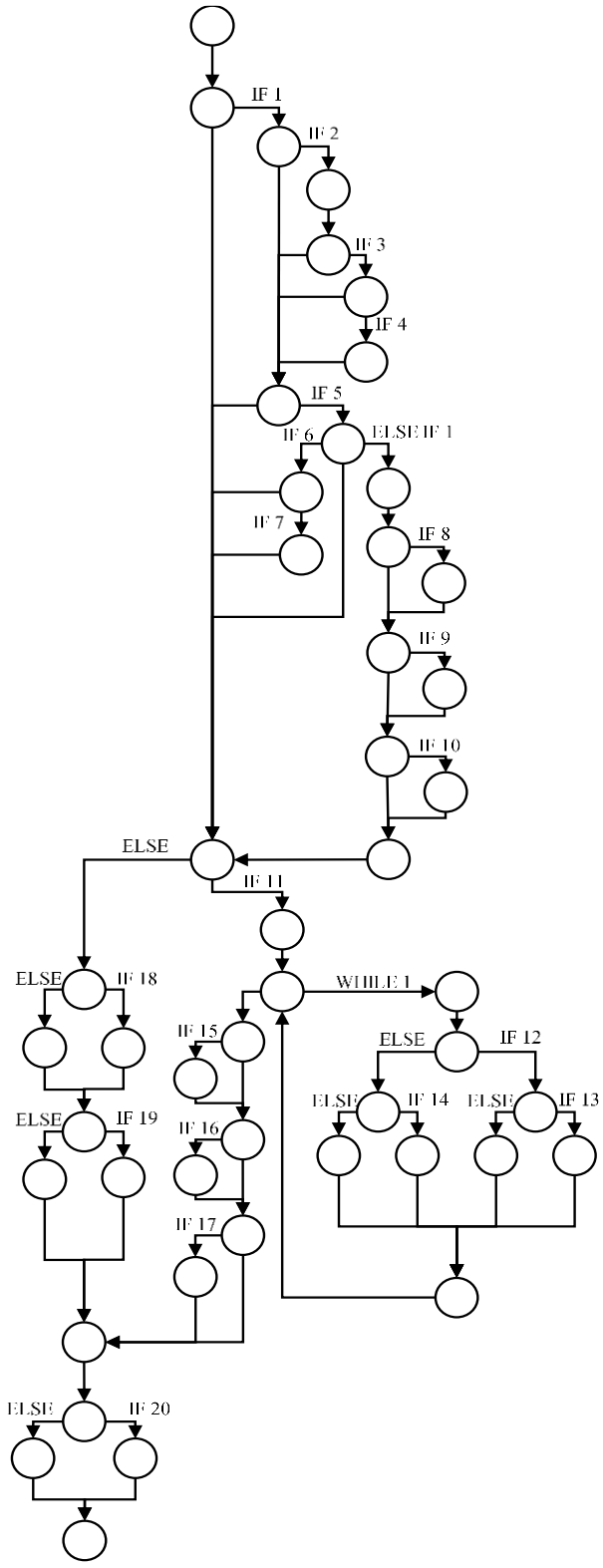


Input variables: val1, val2, val3
FOR 1: for (i = 0; i < 100; i++)
IF 1: (val1 > 5 & val1 < 60) | (val2 > 90 | val2 = 10)
IF 2: val1 = 60
IF 3: val3 > 5 & val1 < 30 & weight_count > 1
IF 4: val3 > 50
IF 5: val1 < val3
WHILE 1: while (n < 10)
IF 6: val1 > 50
FOR 2: for (i = 0; i < output.Count; i++)
IF 7: val1 > 60 & val2 > 55 & weight_count > 1
IF 8: val2 > 90
IF 9: val2 == val3

Рисунок 2.15 – Граф и реализация условий и циклов для второй вариации SUT1. Красным выделены недостижимые операторы

В качестве SUT2 (Рисунок 2.16) используется специально созданная программа с большим количеством операторов и условий. Основные вычисления производятся в функциях, которые не представлены на рисунке, но отражены в приложении А. В дополнение к большему количеству операторов и функций в

SUT2 введена иерархия условий для проверки производительности предложенного метода автоматической генерации тестовых данных.



Input variables: executionNumber, val1, val2, val3, val4, val5
IF 1: IsPalindromeNumber(val1) || IsPalindromeNumber(val2) || IsPalindromeNumber(val3) || IsPalindromeNumber(val4) || IsPalindromeNumber(val5)
IF 2: val1 == Largest4 Number (val2, val3, val4, val5)
IF 3: val2 <= val4/2
IF 4: IsOddNumber(val2) && IsOddNumber(val4)
IF 5: IsPrimeNumber(val1) && IsPrimeNumber(val3) && IsPrimeNumber(val5)
IF 6: NumberMoreThan(val1, val3) && NumberMoreThan(val1, val5))
IF 7: val1 < val2 && val1 < val4
ELSE IF 1: NumberLessThan(val3, val5)
IF 8: val1 > 0
IF 9: val3 > 0
IF 10: val5 > 0
IF 11: val2 > val3 && val1 < val5
WHILE 1: checkDouble <= max
IF 12: val1 < val3
IF 13: checkDouble % 2 == 0
IF 14: IsAmicableNumbers(val1, val3)
IF 15: val3 < val2
IF 16: val1 < val5
IF 17: val3 < val2 && val1 < val5
IF 18: Power(val3, 2) < val2
IF 19: Power(val1, 2) < val5
IF 20: executionNumber < 100)

Рисунок 2.16 – Граф и реализация условий и циклов для SUT2

2.5.2 Поиск тестовых данных для одного пути программного кода

После описания тестовых кодов для проверки алгоритма, проведём исследование алгоритма генерации тестовых данных для одного, наиболее сложного, пути. В качестве тестовой программы используется SUT1, в качестве метрики оценки сложности кода выбрана NOD. Размер популяции и количество поколений $m = Q = 100$. Генерируемые значения ограничены интервалом (0, 100). Таблица 2.5 показывает сгенерированные тестовые наборы.

Таблица 2.5 – Результаты генерации тестовых данных для покрытия одного пути

Поколение	**Запуск 1**	**Запуск 2**	**Запуск 3**	**Запуск 4**
0	1: 15, 67, 26 2: 32, 27, 83 3: 37, 52, 64 4: 70, 49, 64 5: 67, 29, 94	1: 92, 97, 28 2: 38, 66, 52 3: 63, 76, 64 4: 7, 24, 56 5: 57, 48, 8	1: 97, 3, 6 2: 82, 77, 64 3: 24, 47, 57 4: 90, 13, 82 5: 81, 69, 24	1: 78, 23, 35 2: 62, 36, 95 3: 52, 35, 27 4: 17, 77, 73 5: 75, 9, 96
20	1: 99, 71, 45 2: 99, 71, 15 3: 99, 71, 3	1: 99, 13, 10 2: 99, 13, 11 3: 99, 13, 11	1: 97, 80, 4 2: 97, 80, 53 3: 97, 80, 28	1: 95, 64, 54 2: 95, 64, 29 3: 95, 64, 54
50	1: 99, 71, 60 2: 99, 71, 3 3: 99, 71, 3	1: 99, 13, 10 2: 99, 13, 11 3: 99, 13, 11	1: 97, 80, 29 2: 97, 80, 4 3: 97, 80, 53	1: 95, 64, 54 2: 95, 64, 29 3: 95, 64, 54
Итоговое (99)	**1: 99, 71, 60** 2: 99, 71, 45	**1: 99, 13, 10** 2: 99, 13, 11	**1: 97, 80, 4** 2: 97, 80, 29	**1: 95, 64, 54** 2: 95, 64, 29

Всего было инициировано 4 запуска алгоритма, все из которых успешно сгенерировали тестовые наборы для выхода на самый сложный путь уже до 20 поколения. Результатом генерации данных для одного пути является первая (наиболее приспособленная) хромосома последнего поколения. Анализ полученных результатов позволяет отметить закономерности в распределении значений тестовых наборов – первое значение всегда максимальное и находится

около верхней границы ограничения, второе меньше первого, но больше третьего. Можно заметить увеличение количества неразличимых хромосом с увеличением числа пройденных поколений, что позволяет сделать вывод о невозможности генерировать данные для множества путей одним запуском.

2.5.3 Генерация данных многократным запуском генетического алгоритма

Для достижения максимального покрытия можно использовать многократный запуск алгоритма генерации тестовых данных для одного пути, тем самым последовательно увеличивая значение покрытия наборами, проходящим по отличным путям. Для обеспечения работоспособности данного метода необходимо дополнительно встроить механизм исключения уже покрытых операторов при каждом последующем запуске. Для реализации данного механизма решение одного запуска и путь, который инициирован сгенерированным набором, сохраняются, после чего для покрытых операторов j индикатор покрытия g_j фиксируется на значении 0 с невозможностью его дальнейшего изменения, что исключает их влияние на значение функции приспособленности в последующих запусках.

Для исследования данного метода используется вторая вариация SUT1, которая не позволяет достичь полного покрытия всех операторов. Использование второй вариации используется для отслеживания невозможности получения новых тестовых наборов при достижении определённого значения покрытия, что позволит сделать вывод о возможности применения алгоритма для выявления в тестовом коде недостижимых частей.

Для представления результатов выводится номер итерации, сгенерированный набор данных, функция приспособленности для данного набора и итоговое покрытие кода (Таблица 2.6). В качестве метрики оценки сложности кода используется метрика NOD, и не были использованы методы смешивания.

Алгоритм сгенерировал восемь тестовых наборов данных, шесть из которых имеют ненулевое значение функции приспособленности, то есть покрывают

определённые пути кода. Итоговое покрытие кода составило 89%, после достижения которого алгоритм начал генерировать тестовые наборы с нулевым значением функции приспособленности. Из-за наличия недостижимых условий во второй вариации SUT1, значение покрытие операторов 89% является полным, то есть алгоритм сгенерировал 6 тестовых вариантов для прохождения по всем возможным операторам кода.

Таблица 2.6 – Результаты многократного запуска алгоритма генерации тестовых данных для одного пути

Запуск ГА	**Набор данных**	**Значение функции приспособленности**	**Покрытие кода**
1	(77, 18, 99)	164 100	41%
2	(60, 41, 61)	26 400	53%
3	(5, 8, 56)	18 882	74%
4	(40, 30, 55)	9 900	79%
5	(99, 73, 73)	12 000	86%
6	*(84, 22, 64)*	*5 000*	*89%*
7	(48, 72, 44)	0	89%
8	(36, 2, 36)	0	89%

С каждой новой итерацией значение функции приспособленности постепенно снижается, то есть каждый запуск покрывает все меньшее число новых операторов. Исключением является 5-ая итерация, когда значение функции выше, чем в предыдущем. Это говорит о том, что выход на путь, покрытый пятым набором данных, ограничен более сильными условиями. Также возможна ситуация, когда алгоритм при случайной генерации получил данные для выхода на менее сложный путь, и, при текущих параметрах ГА, ему не хватило времени для поиска тестовых наборов для выхода на потенциально сложный путь. В таком случае возможно установка других параметров ГА, чтобы дать алгоритму больше времени на поиск данных для других путей.

Выводы по главе 2

В данной главе исследуется применение генетического алгоритма для задачи генерации тестовых данных. Были получены следующие результаты:

1. Адаптированы основные понятия генетического алгоритма, включающие в себя ген, хромосому и популяцию, для решения задачи генерации тестовых данных. Обосновано применение непрерывного варианта ГА, определено понятие неразличимых хромосом, рассмотрены особенности реализации эволюционных операций применительно к решаемой задачи.

2. Определены основные этапы генетического алгоритма. Предложен новый вариант операции отбора, основанный на выборе лучших хромосом, случайного определения пар и турнирного отбора. Для обеспечения большего разнообразия популяции представлена эволюционная операция смешивания, позволяющая вводить в популяцию новые значения.

3. Предложено использовать метрики оценки сложности кода SLOC, ABC и Джилба для вычисления весов функции приспособленности. Предложена метрика NOD, позволяющая учитывать уровень вложенности операторов при генерации тестовых данных, а также учитывающей количество выполнений операторов. Метрики SLOC и NOD были выбраны в качестве основных для проведения дальнейших исследований.

4. Разработан алгоритм генерации тестовых данных для одного сложного пути программного кода, который определяется весами операторов, находящихся на нём. Было выявлено, что при использовании данного метода формируется большое число неразличимых хромосом, что говорит о его неэффективности при формировании тестовых наборов.

5. Исследован алгоритм многократного запуска алгоритма генерации тестовых данных для одного пути с целью достижения полного покрытия тестируемой программы. В отличие от существующих методов, алгоритм позволяет сгенерировать необходимое количество тестовых наборов для достижения необходимого значения покрытия.

ГЛАВА 3 ГЕНЕРАЦИЯ НАБОРОВ ТЕСТОВЫХ ДАННЫХ ДЛЯ ОБЕСПЕЧЕНИЯ МАКСИМАЛЬНОГО ПОКРЫТИЯ КОДА

В предыдущей главе рассмотрена формальная постановка задачи генерации тестовых данных для ее решения с помощью генетического алгоритма. Показано, что качество сгенерированных данных, определяемое степенью покрытия тестируемого кода, зависит как от конкретных вариантов выбора параметров генетических операций, так и, в преобладающей степени, от весовых коэффициентов функции приспособленности, которые предложено определять на основе метрик оценки сложности кода.

На основе формальной постановки задачи была предложена реализация ГА для генерации одного тестового набора, покрывающего самый сложный путь на графе потоков управления. Исследования установили, что в результате одного запуска ГА с предложенной в разделе 2 функцией приспособленности получается вырожденная популяция тестовых данных, содержащая большое множество неразличимых хромосом, покрывающих сравнительно небольшое число самых сложных путей тестируемого кода, или даже выводящих на один и тот же путь. Для применения данной реализации ГА к решению задачи нахождения множества тестовых наборов, покрывающих как можно большее число путей (по возможности, весь анализируемый код), предложен вариант многократного запуска ГА, где каждый следующий запуск проводится после обнуления весов ранее покрытых операторов. Однако такой способ нельзя считать эффективным, так как он является слишком затратным и избыточным. В худшем случае для покрытия каждого следующего пути необходим новый запуск ГА, состоящий из множества итераций.

В данной главе предлагаются две модификации разработанного ранее подхода, позволяющие сгенерировать максимально разнообразную популяцию наборов тестовых данных в рамках одного запуска ГА. Обе модификации основаны на гипотезе о возможности увеличения разнообразия популяции из поколения в

поколение (от одной итерации к другой) путем выбора соответствующего вида функции приспособленности ГА. В первой модификации в функцию приспособленности вводится дополнительная аддитивная компонента, отвечающая за разнообразие популяции. Во второй модификации мы ограничиваемся одной компонентой функции приспособленности, отвечающей за сложность путей кода, но вводим основанный на идее муравьиного алгоритма механизм динамического изменения весов данной функции. Результаты исследований, рассмотренных в данной главе, были опубликованы в [1–6, 15, 16].

3.1 Модификация функции приспособленности на основе введения аддитивной компоненты, отвечающей за разнообразие популяции

Во второй главе была определена функция приспособленности ГА, учитывающая сложность пути на основе весов соответствующих операторов

$$\mathrm{F}_1(x_i) = \sum_{j=1}^{n} w_j g_j(x_i), \tag{3.1}$$

где $g(x_i) = \big(g_1(x_i), g_2(x_i), \dots, g_n(x_i)\big)$ – вектор, являющийся индикатором покрытия операторов, инициированным тестовым набором x_i.

Чтобы обеспечить большее разнообразие популяции, в формулу (3.1) добавляется компонента, которая позволяет учитывать удаленность путей друг от друга. Удаленность путей определим через операцию схожести. Для вычисления j-го коэффициента схожести $sim_j\big(x_{i_1}, x_{i_2}\big)$ двух хромосом x_{i_1} и x_{i_2}, проверяем, находится ли j-й оператор тестируемого кода, покрытие которого маркируется индикатором g_j, на пересечении обоих путей, инициированных тестовыми наборами x_{i_1} и x_{i_2}:

$$sim_j\big(x_{i_1}, x_{i_2}\big) = \overline{g_j\big(x_{i_1}\big) \oplus g_j\big(x_{i_2}\big)}, j = \overline{1, n}, \tag{3.2}$$

где использованы логические операции «отрицание» (NOT) и $\oplus$ – «исключающее или» (XOR).

Чем больше совпадающих покрытых операторов на пересечении двух путей, тем больше значение сходства между хромосомами. Следующая формула задает схожесть между двумя хромосомами как средневзвешенное значение схожести по всем операторам кода:

$$sim(x_{i_1}, x_{i_2}) = \sum_{j=1}^{n} w_j \cdot sim_j(x_{i_1}, x_{i_2}). \quad (3.3)$$

Значение схожести между хромосомой x_i и остальными хромосомами популяции вычисляется как

$$f_{sim}(x_i) = \frac{1}{(m-1)} \sum_{\substack{s=1 \\ s \neq i}}^{m} sim(x_s, x_i), \quad (3.4)$$

где m – число хромосом в популяции.

Теперь можно определить среднее значение сходства путей во всей популяции

$$\overline{f_{sim}} = \frac{1}{m} \sum_{i=1}^{m} f_{sim}(x_i). \quad (3.5)$$

и далее сформулировать аддитивную компоненту функции приспособленности, ответственную за разнообразие путей в популяции, как модуль разности между средней схожестью популяции и схожестью конкретной хромосомы

$$F_2(x_i) = \left| \overline{f_{sim}} - f_{sim}(x_i) \right|. \quad (3.6)$$

В итоге, результирующая функция приспособленности для хромосомы x_i с учётом схожести вычисляется по формуле

$$F(x_i) = F_1(x_i) + k \cdot F_2(x_i), \quad (3.7)$$

где компоненты $F_1(x_i)$ и $F_2(x_i)$ определяются формулами (3.1) и (3.6). Соответственно, первая компонента $F_1(x_i)$ определяет сложность пути, инициированного хромосомой x_i, а вторая компонента $F_2(x_i)$ – удалённость этого

пути от всех остальных путей в популяции. Параметр k определяет соотношение между ними.

Использование формулы (3.7) в качестве функции приспособленности приводит к более разнообразным популяциям в результате одного запуска ГА. Тем не менее, в силу использования в данной работе непрерывной версии генетического алгоритма (даже с использованием продвинутых механизмов смешивания, увеличивающих влияние случайного фактора), полученное разнообразие недостаточно для полного покрытия кода в рамках одного запуска ГА.

Последнее обстоятельство связано с выявленным в процессе исследования «эффектом раскачивания», возникающим из-за наличия в популяции неразличимых хромосом. Если неразличимые хромосомы будут иметь высокое значение функции приспособленности в одном поколении, то они будут отобраны для скрещивания. Тогда их потомки, с высокой вероятностью, также будут неразличимыми с родителями. Новое поколение, в таком случае, будет состоять из большего числа неразличимых хромосом и схожесть в популяции будет зависеть в большей степени от них. Для всех этих хромосом значение аддитивной компоненты функции приспособленности F_2 будет снижено, а для хромосом, проходящих по другим путям, оно будет увеличено. Теперь уже другие хромосомы могут быть выбраны для скрещивания и будут формировать для следующего поколения множество неразличимых хромосом. Таким образом, популяция будет циклически сначала наполняться неразличимыми хромосомами, что в следующем поколении приведет к снижению для них значения сходства, а, соответственно, и к снижению F_2, тем самым уменьшая приоритет неразличимых хромосом на следующей итерации. Подобный цикл будет повторяться для разных наборов, и в результате сложность пути F_1 и значение сходства F_2 перестают играть важной роли при формировании новой популяции, а разные наборы постоянно перетасовываются без проникновения вглубь пространства решений.

Для исключения данного эффекта раскачивания, предлагается использовать значение $ind(x_1, \dots, x_i)$, которое определяется числом хромосом из множества $\{x_1, \dots, x_{i-1}\}$ неразличимых с хромосомой x_i:

$$\tilde{F}(x_i) = F_1(x_i) + \frac{k}{1 + ind(x_1, \dots, x_i)} \cdot F_2(x_i). \quad (3.8)$$

Действительно, начальное значение $ind(x_1) = 0$, так как множество, в котором выявляются неразличимые хромосомы, пусто на первом шаге. На каждом следующем шаге значение $ind(x_1, \dots, x_i)$ может либо увеличиваться на 1, если следующая хромосома неразличима с одной из предшествующих, либо сохранять прежнее значение, если следующая хромосома уникальна. Это позволит хромосомам, проходящим по разным путям, более равномерно распределиться по популяции в целом.

3.2 Исследование модификации функции приспособленности

Для исследования возможностей ГА генерировать тестовые наборы с предложенной функцией приспособленности (3.8), будут использованы обе тестовые программы, предложенные в п. 2.5.1, то есть программы SUT1 и SUT2.

3.2.1 Анализ быстродействия алгоритма генерации данных при изменении количества хромосом и поколений

Применение генетического алгоритма часто связано с проблемой выбора подходящих параметров, а именно, количества поколений (Q) и размера популяции (m). Это осложняется тем, что для больших полносвязных сложнологических программ оценить возможное влияние изменения одного из параметров достаточно трудно, а предугадать поведение алгоритма при определённых параметрах может стать отдельной проблемой.

Для того чтобы оценить эффект изменения параметров ГА, было проведено исследование быстродействия алгоритма генерации одного тестового набора для SUT1 с постепенным увеличением одного из параметров при сохранении значения

другого. На графиках ниже (Рисунок 3.1 и Рисунок 3.2) можно отследить, каким образом изменяется скорость работы приложения. Линиями показывается абсолютное время выполнения программы в секундах, столбиками – разность времени работы алгоритма по сравнению с предыдущим значением. При исследовании одного из параметров, второй приравнивался к постоянному значению 100. Рисунок 3.1 позволяет оценить влияние размера популяции на время работы алгоритма.

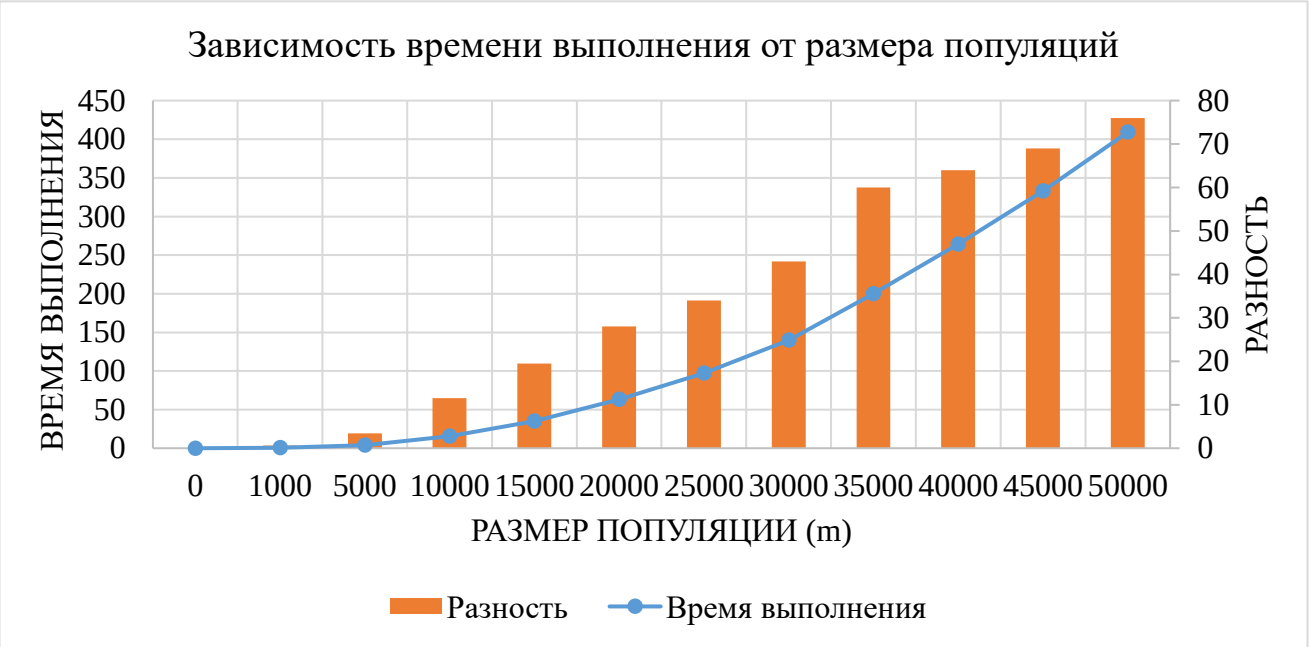


Рисунок 3.1 – Зависимость длительности генерации данных от количества хромосом в популяции

Из-за малого размера SUT1 скорость генерации одного набора данных достаточно высокая, поэтому для исследования было решено существенно увеличить размер популяции до 50 000 хромосом. Хотя такое количество существенно выше необходимого, благодаря данному исследованию стало возможным увидеть нелинейную тенденцию к увеличению длительности работы при увеличении числа хромосом. Это позволяет сделать вывод о существенном влиянии размера популяции на быстродействие алгоритма. С другой стороны,

размер популяции является важным параметром для обеспечения достаточно высокого уровня разнообразия тестовых наборов.

Теперь рассмотрим зависимость скорости работы алгоритма от числа поколений (Рисунок 3.2). При изменении числа поколений наблюдается линейная тенденция, поэтому очевидно, что изменение количества поколений оказывает меньшее влияние на общее быстродействие алгоритма. При изменении размера популяции длительность работы алгоритма увеличилась до 10 минут, тогда как при изменении числа поколений она достигла лишь нескольких секунд.

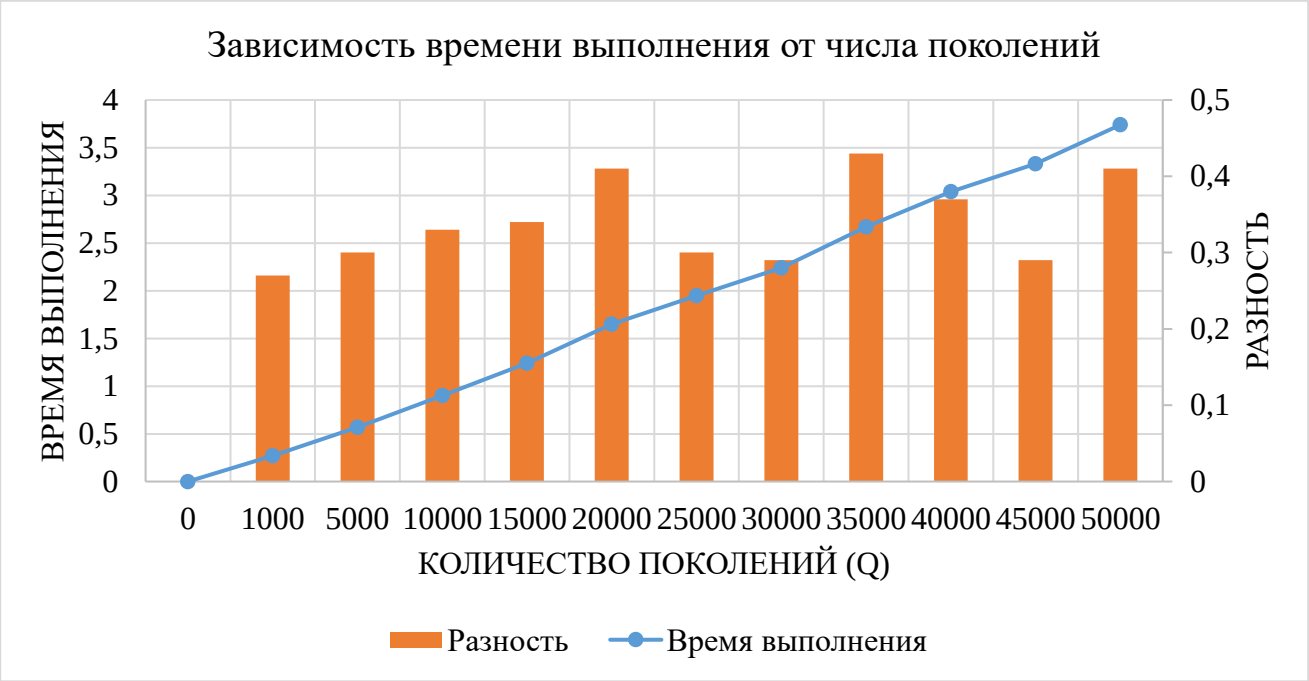


Рисунок 3.2 – Зависимость длительности генерации данных от количества поколений

Несмотря на то, что и в одном, и в другом случае общее количество хромосом и поколений оставалось неизменным, наблюдаются существенные различия в степени влияния параметров ГА, а именно, количества поколений (Q) и размера популяции (m), на время работы алгоритма. Увеличение размера популяции существенно замедляет скорость работы алгоритма – число хромосом в популяции с 1 000 до 50 000 увеличилось в 50 раз, но длительность при этом возросла в 750.

Увеличение же количества поколений в 50 раз привело к увеличению времени лишь в 14 раз.

Данный эффект можно объяснить тем, что наиболее вычислительно сложные эволюционные операции происходят при вычислении функции приспособленности и при поиске оптимальных для скрещивания хромосом. Именно из-за того, что поиск лучших хромосом зависит от числа хромосом в популяции, то при сортировке значений и отборе лучших хромосом длительность работы существенно возрастает. При увеличении количества поколений возрастает лишь количество итераций выполнений, но не сложность внутри итерации.

Важно также отметить, что размер популяции оказывает существенное влияние на разнообразие тестовых вариантов, а обеспечение достаточно высокого уровня разнообразия является одним из основных требований к успешной работе ГА. Изменение количества популяций имеет гораздо меньшее влияние на работу алгоритма, так как лишь увеличивает количество выполнений в надежде, что при заданном размере популяции решение удастся найти. Можно сказать, что число хромосом является качественным показателем решения, а число поколений – количественным.

Поэтому при подборе параметров генетического алгоритма приоритет стоит отдавать именно размеру популяции, соблюдая баланс между скоростью работы алгоритма и качеством полученных результатов. Действительно, слишком малый размер популяции неспособен обеспечить достаточное разнообразие итоговой популяции и, соответственно, степень покрытия кода.

Для повышения разнообразия также могут быть использованы другие механизмы генетического алгоритма. В рамках данных работы для увеличения разнообразия популяции и устранения негативного влияния использования непрерывного ГА предлагается использовать эволюционную операцию смешивания, о чем будет сказано в следующем разделе.

3.2.2 Исследование влияния методов смешивания на покрытие кода

Смешивание используется для введения в популяцию новых значений, которые не могут быть получены при использовании скрещивания в непрерывном ГА. Подробно методы смешивания были описаны в п. 2.3.3.

Для исследования методов смешивания будет использована первая вариация тестовой программы SUT1, параметры ГА эмпирически подобраны таким образом, чтобы достижение полного покрытия не гарантировалось в последней популяции – количество поколений $Q = 50$, размер популяции $m = 25$.

Для вычисления среднего значения покрытия было произведено 50 запусков алгоритма. В таблице 3.1 представлены результаты использования разных методов смешивания, среднее значение покрытия и доля запусков, в которых достигнуто полное покрытие.

Таблица 3.1 – Результаты применения методов смешивания

Метод смешивания	**Среднее значение покрытия**	**Доля запусков с полным покрытием**
Без смешивания	79%	12%
Прямое смешивание	75%	16%
Усредненное смешивание	66%	4%
Сдвигающееся смешивание	77%	16%

Методы смешивания показали худшие результаты по среднему значению покрытия, но при этом методы прямого и сдвигающегося смешивания чаще достигают полного покрытия тестового кода. Худшие результаты были получены при использовании простого усреднения значений родительских генов.

Более частое достижение полного покрытия методами прямого и сдвигающегося смешивания говорит о достижении большего разнообразия популяции, хоть и ценой снижения среднего покрытия. Таким образом, применение данных методов смешивания подходит для тестовых программ с

большим количеством путей – чем больше «разветвляется» программа, тем важнее становится обеспечение достаточно высокого уровня разнообразия.

Поэтому в качестве основного метода смешивания в работе используется сдвигающееся смешивание, так как несущественное снижение в покрытии может быть нивелировано другими механизмами, а дополнительное разнообразие значений тестовых данных имеет более важное значение.

3.2.3 Определение параметров алгоритма для достижения полного покрытия

В ходе исследования были рассмотрены два варианта алгоритма генерации тестовых данных – многократный запуск ГА и модификация с использованием аддитивной компоненты F_2. Для вычисления функции приспособленности используется формула (3.7). Исследования проведены для различных значений параметров ГА - размера популяции (m) и количества поколений (Q) для шести сочетаний данных параметров.

Результаты использования алгоритма многократного запуска ГА представлены в таблице 3.2. Можно заметить, что при слишком маленьком размере популяции ($m = 5$) встречаются «холостые» запуски алгоритма, которые не обеспечивают дополнительного покрытия по сравнению с предшествующим запуском (такие тестовые наборы и соответствующие им векторы $g(x)$ индикаторов покрытия выделены курсивом). Несмотря на это, в итоге, в результате нескольких запусков, достигается полное покрытие кода.

Однако, как уже отмечалось ранее, многократный запуск ГА не является эффективным методом, поэтому желательно достигнуть полного покрытия при одном запуске алгоритма с модифицированной функцией приспособленности, обеспечивающей повышение разнообразия популяции от поколения к поколению.

Таблица 3.2 – Результаты многократного запуска ГА с первоначальной функцией приспособленности (3.1)

Число поколений, Q	**Размер популяции, m**	**Тестовый набор, x**	**Покрытие, %**	**Индикатор покрытия операторов $g(x)$**
10	5	(6, 91, 20)	27%	1111111111111000000000000000000000000000000000000001
		(86, 41, 98)	58%	1111111111100000000000000001111111111111111000000001
		(29, 99, 83)	*58%*	*1111111111111000000000000000000000000000000000000001*
		(51, 48, 3)	*58%*	*1111111111111000000000000000000000000000000000000001*
		(60, 9, 60)	72%	1111111111100011111111000000000000000000000000000001
		(2, 42, 80)	100%	1111111111000000000001111111111111000000001111111111
	10	(61, 80, 70)	52%	1111111111100000000000000001111111111111111000000001
		(60, 15, 55)	67%	1111111111100011111111000000000000000000000000000001
		(57, 16, 74)	72%	1111111111111000000000000000000000000000000000000001
		(3, 54, 85)	100%	1111111111000000000001111111111111000000001111111111
	20	(72, 40, 96)	52%	1111111111100000000000000001111111111111111000000001
		(3, 72, 69)	80%	1111111111000000000001111111111111000000001111111111
		(38, 83, 81)	85%	1111111111111000000000000000000000000000000000000001
		(60, 49, 35)	100%	1111111111100011111111000000000000000000000000000001
20	5	(67, 35, 84)	52%	1111111111100000000000000001111111111111111000000001
		(98, 92, 93)	58%	1111111111111000000000000000000000000000000000000001
		(93, 13, 78)	*58%*	*1111111111100000000000000000000000000000000000000000*
		(60, 30, 15)	72%	1111111111100011111111000000000000000000000000000001
		(3, 30, 72)	100%	1111111111000000000001111111111111000000001111111111
	10	(69, 85, 72)	52%	1111111111100000000000000001111111111111111000000001
		(60, 41, 8)	67%	1111111111100011111111000000000000000000000000000001
		(2, 69, 64)	94%	1111111111000000000001111111111111000000001111111111
		(7, 90, 41)	100%	1111111111111000000000000000000000000000000000000001
	20	(83, 40, 98)	52%	1111111111100000000000000001111111111111111000000001
		(60, 7, 44)	67%	1111111111100011111111000000000000000000000000000001
		(54, 6, 65)	72%	1111111111111000000000000000000000000000000000000001
		(4, 11, 94)	100%	1111111111000000000001111111111111000000001111111111

Таблица 3.3 показывает результаты работы алгоритма для модифицированной функции приспособленности вида (3.7) для $k = 10$. Результаты показывают, что с использованием данного вида функции удается достичь полного покрытия 100%, при однократном запуске ГА, однако это выполняется лишь при достижении минимального размера популяции (для данного

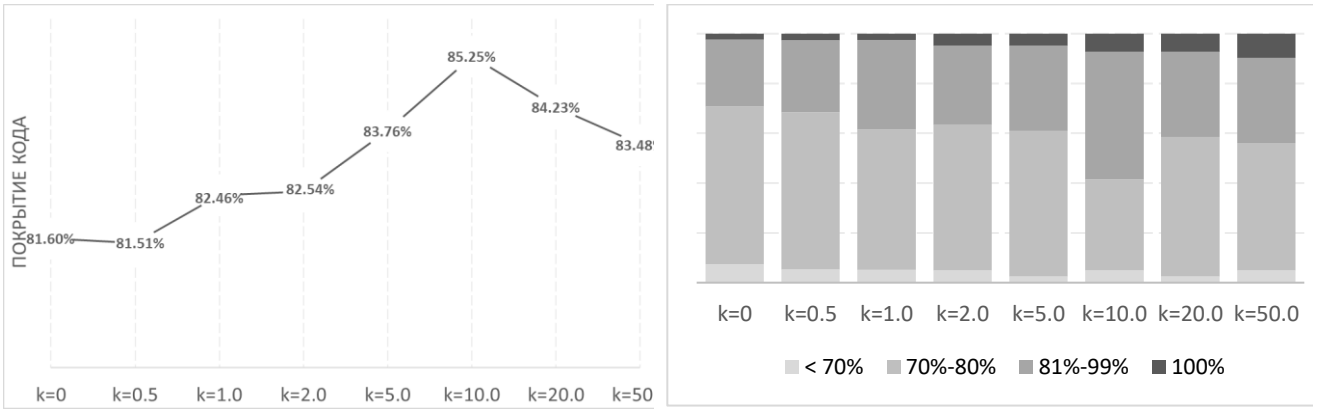
тестового кода $m = 20$) и минимального числа поколений (для данного тестового кода $Q = 20$).

Таблица 3.3 – Результаты однократного запуска ГА с модифицированной функцией приспособленности (3.8)

Число поколений, Q	**Размер популяции, m**	**Тестовый набор, x**	**Покрытие, %**	**Индикатор покрытия операторов $g(x)$**
10	5	(60, 41, 43)	72%	11111111111000111111110000000000000000000000000000000001
		(70, 81, 86)		11111111111000000000000000011111111111111111110000000001
		(28, 27, 67)		11111111111111000000000000000000000000000000000000000001
		Итог		11111111111111111111110000001111111111111111110000000001
	10	(30, 5, 8)	72%	11111111111111000000000000000000000000000000000000000001
		(83, 39, 84)		11111111111000000000000000011111111111111111110000000001
		(60, 51, 67)		11111111111000111111110000000000000000000000000000000001
		Итог		11111111111111111111110000001111111111111111110000000001
	20	(61, 88, 59)	85%	11111111111000000000000000000000000000000000000000000000
		(52, 81, 42)		11111111111111000000000000000000000000000000000000000001
		(62, 89, 80)		11111111111000000000000000011111111111111111110000000001
		(2, 49, 74)		11111111111000000000001111111111111110000000001111111111
		Итог		11111111111111000000001111111111111111111111111111111111
20	5	(33, 17, 94)	72%	11111111111111000000000000000000000000000000000000000001
		(67, 8, 95)		11111111111000000000000000011111111111111111110000000001
		(60, 33, 105)		11111111111000111111110000000000000000000000000000000001
		Итог		11111111111111111111110000001111111111111111110000000001
	10	(65, 49, 90)	85%	11111111111000000000000000011111111111111111110000000001
		(3, 1, 57)		11111111111000000000001111111111111110000000001111111111
		(37, 2, 57)		11111111111111000000000000000000000000000000000000000001
		Итог		11111111111111000000001111111111111111111111111111111111
	20	(37, 17, 87)	100%	11111111111111000000000000000000000000000000000000000001
		(0, 70, 86)		11111111111000000000001111111111111110000000001111111111
		(78, 0, 90)		11111111111000000000000000011111111111111111110000000001
		(60, 41, 71)		11111111111000111111110000000000000000000000000000000001
		Итог		11111111111111111111111111111111111111111111111111111111

Для изучения влияния константы k в формуле (3.7) на процесс генерации тестовых наборов были подобраны параметры размера популяции и количества поколений, $m = Q = 35$, при которых полное покрытие операторов первой вариации SUT1 достигается довольно редко. Рисунок 3.3а показывает среднее значение достигнутого покрытия операторов для различных значений k от 0 до 50,

вычисленное для 40 независимых запусков ГА. Рисунок 3.3б отражает соотношение между группами запусков ГА с различным достигнутым покрытием при различных значениях k, что позволяет в лучшей степени отразить разнообразие генерируемых наборов.



(а) Среднее значение покрытия (б) Группы с различным покрытием

Рисунок 3.3 – Достигнутое покрытие операторов в последнем поколении при $(m = Q = 35)$

Таким образом, можно отметить нелинейную зависимость среднего значения покрытия от параметра k, определяющего соотношение между компонентами F_1 и F_2 функции приспособленности F в (3.7). Во-первых, с увеличением k значение покрытия операторов увеличивается, достигая максимального значения при $k = 10$. При дальнейшем увеличении k компонента F_2, отвечающая за разнообразие, начинает оказывать большее влияние на значение функции приспособленности, из-за чего снижается важность компоненты F_1, отвечающей за сложность пути. По этой причине, тестовые наборы перестают эволюционировать с каждым следующим поколением, а начинают просто перемешиваться, результатом чего является снижение итогового покрытия.

Также интересно рассмотреть распределение групп с различным покрытием кода (Рисунок 3.3б). Можно отметить, при небольшом значении k алгоритм намного чаще генерирует тестовые наборы, покрывающие от 70% до 80% тестового кода. При $k = 10$ наибольшая доля приходится на группу с высоким

покрытием 81%–99%. В дальнейшем, для $k = 20$ и $k = 50$ распределение в целом ухудшается по сравнению с $k = 10$ (хотя доля вариантов с полным покрытием выше, доля с низким покрытием также увеличивается).

Таким образом, можно сделать вывод, что существует определенное значение k, при котором достигается баланс между компонентами F_1 и F_2 (для исследуемого кода $k = 10$). Если F_1 имеет преобладающее влияние, то результирующая популяция будет недостаточно разнообразна, чтобы покрыть множество различных путей. Если же преобладает F_2, то чрезмерное влияние оказывает разнообразие популяции в ущерб качеству покрытию каждой особи в отдельности.

3.2.4 Исследование соотношения компонент функции приспособленности на процесс генерации тестовых данных

В данном разделе исследуем влияние соотношения между компонентами функции приспособленности F_1 и F_2 , определяемое параметром k, на качество покрытия кода. Использование для той цели тестовой программы SUT2, содержащей большое число условных операторов, и, соответственно, большое число путей, позволит более детально исследовать этот вопрос. В отличие от исследований в предыдущем разделе, здесь мы будем использовать усовершенствованную формулу (3.8) для вычисления функции приспособленности, учитывающую наличие неразличимых хромосом в популяции с помощью параметра $ind(x_1, \dots, x_i)$.

На рисунке 3.4 показаны результаты исследования для различных размеров популяции m, и различных значений параметра k. Чтобы провести более детальный сравнительный анализ, мы ограничились значением числа поколений $Q = 75$, при котором полное покрытие не было достигнуто.

Во-первых, отметим, что введение параметра $ind(x_1, \dots, x_i)$ существенно улучшило результаты покрытия, несмотря на большую сложность тестируемого

кода. В целом, зависимость степени покрытия от параметра k для кода SUT2 напоминает соответствующую зависимость для SUT1, с максимумом в средней точке $k = 10$, хотя этот максимум выражен в меньшей степени. Мы объясняем это введением в функцию приспособленности параметра $ind(x_1, \dots, x_i)$, что позволило существенно увеличить среднее значение степени покрытия и, дополнительно к этому, снизить влияние параметра k на конечный результат.

Тем не менее, наилучшее значение среднего покрытия всё равно достигается при $k = 10$, хотя тестируемые коды SUT1 и SUT2 существенно различаются по количеству и сложности операторов. Таким образом, значение $k = 10$ можно предположительно рекомендовать в качестве отправной точки при применении функции приспособленности вида (3.8) и для других тестируемых кодов.

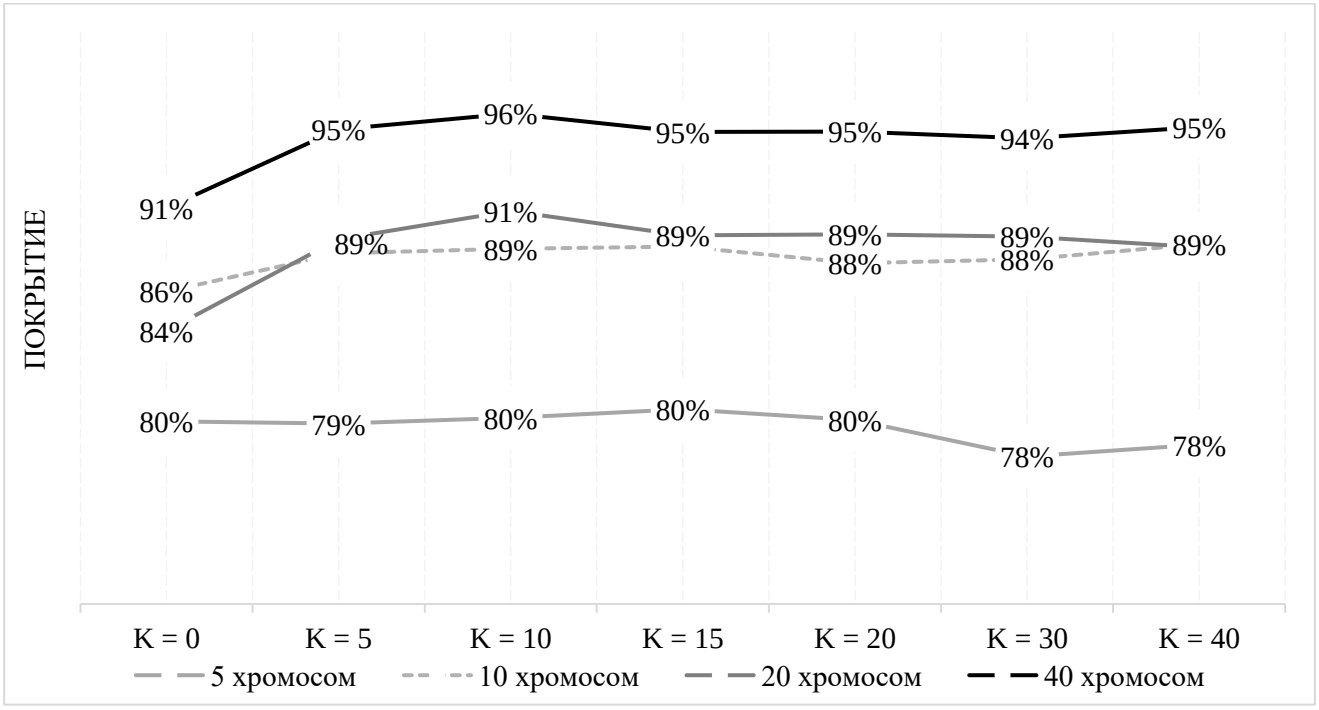


Рисунок 3.4 – Среднее покрытие операторов в зависимости от k для SUT2

Для тестируемого кода SUT2 было также проведено исследование зависимости среднего значения покрытия операторов от количества поколений Q. Среднее значение покрытия было рассчитано для 10 запусков алгоритма с

функцией приспособленности (3.8) с $k = 10$. Результаты представлены для популяции различного размера ($m = 5, 10, 15, 20$ и 40) на рисунке 3.5 хорошо видно, насколько отличается скорость сходимости алгоритма при использовании разного количества хромосом в популяции.

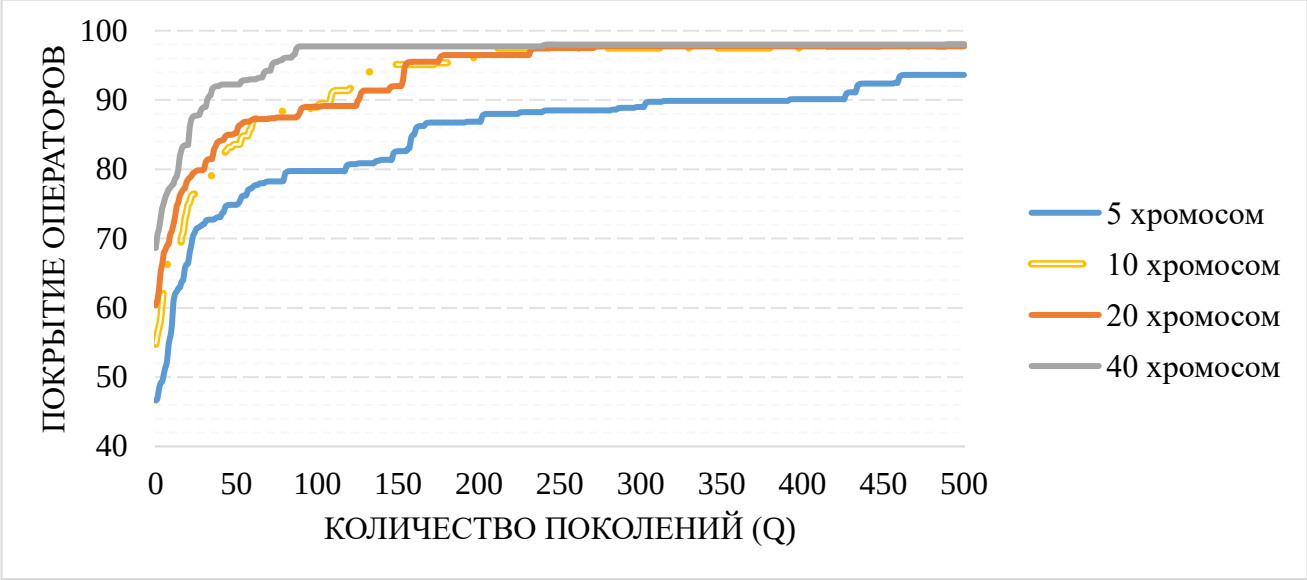


Рисунок 3.5 – Сравнение скорости сходимости алгоритма при различном размере популяции

Прежде всего, следует отметить, что во всех случаях предложенный алгоритм стабильно обеспечивает практически полное покрытие кода (в среднем не менее 98%), хотя скорость сходимости к этому результату сильно зависит от размера популяции. Например, для $m = 5$ полное покрытие обеспечивается приблизительно за 700 поколений. Несколько лучшие результаты показывает алгоритм генерации тестовых данных при размере популяции $m = 10$ и $m = 20$. Максимального покрытия алгоритм достигает при данных параметрах на 270–450 поколениях, т.е. скорость сходимости уже намного выше. Можно отметить, что $m = 20$ является верхней границей количества тестовых наборов, необходимых для обеспечения полного покрытия SUT2. Лучших значений среднего покрытия и скорости сходимости алгоритм достигает при $m = 40$. Максимальное значение покрытия операторов достигается в пределах 70–80 поколений.

Для SUT2 минимальный размер множества сгенерированных тестовых наборов, при котором было достигнуто полное покрытие, составляет приблизительно 15–20. Соответственно, для увеличения скорости достижения полного покрытия при генерации тестовых данных необходимо иметь как минимум в 2 раза больший размер популяции (в нашем исследовании $m = 40$), чем требуемых тестовых наборов для полного покрытия. Дальнейшее увеличение количества хромосом в популяции приводит к увеличению скорости сходимости, но отрицательно сказывается на быстродействии метода, поскольку приводит к чрезмерному увеличению объема вычислений, что было показано в п. 3.2.1.

3.3 Модификация функции приспособленности на основе динамического изменения весов операторов

Исследование, проведённое в п. 3.2.4 показало относительно сильное влияние значения k на покрытие тестируемого кода. При $k = 0$ покрытие было минимальным, достигая своего максимального значения при $k = 10$, после чего начиналось снижение. Очевидно, что выбор правильного k может существенно сказаться на итоговых результатах. Полученное в исследованиях значение $k = 10$ является оптимальным только в рамках протестированных SUT1 и SUT2, для других тестовых программ данное значение может оказаться неоптимальным. Поэтому, с целью достижения большей универсальности алгоритма, было бы предпочтительно уменьшить влияние k.

Функция приспособленности формируется из двух компонент – F_1 и F_2, как показано в формулах (3.7) и (3.8). Компонента F_2, отвечающая за удалённость путей друг от друга, исследована в предшествующих разделах. Она действительно оказывает большое влияние на разнообразие популяции, а, следовательно, и на степень покрытия кода. В данном разделе исследуем, можно ли подобного эффекта разнообразия популяции добиться за счет модификации компоненты F_1. Если это

удастся сделать, то это бы уменьшило влияние компонты F_2, а, следовательно, и влияние константы k в формулах (3.7) и (3.8).

Идея модификации компоненты F_1 заключается в обращении к другим эволюционным подходам. В п. 0 были частично рассмотрены некоторые из эволюционных методов, в частности PSO, но их применение в существующих исследованиях больше базировалось на сравнении реализации PSO и ГА. PSO является одним из алгоритмов Роевого Интеллекта (РИ). Другими представителями этого семейства являются муравьиный алгоритм (Ant Colony Optimization, ACO), пчелиный алгоритм (Artificial Bee Colony Algorithm, ABC), алгоритм кукушки (Cuckoo Search, CS) и множество других алгоритмов, в основе которых находится коллективное взаимодействие различных элементов или агентов.

Муравьиный алгоритм (ACO) [110] является одним из алгоритмов, который позволяет решать задачи по нахождению маршрута на графах. В его основе лежит симуляция поведения колонии муравьев. Муравьи, проходя по определённым путям, оставляют за собой след из феромонов. Чем лучшее решение было найдено, тем больше феромонов будет на том или ином пути. В следующем поколении муравьи уже формируют свои пути на основе количества феромонов – чем больше феромонов на определённом пути, тем больше муравьёв будут направлены на данный путь и продолжат его исследование. Таким образом колония постепенно исследует всё пространство решений, постепенно выходя на всё лучшие и лучшие пути.

Непосредственно применение муравьиного алгоритма не представляется возможным, так как выход на определённые пути инициируется разными наборами данных, и единственный способ изменить путь – это манипуляции исходными данными. Тем не менее, идея использовать «феромоны» для приоритизации поиска путей может оказать положительный эффект для обеспечения большего разнообразия популяции.

Применительно к задаче повышения разнообразия тестовых наборов, инициируемых наборами тестовых данных, идея феромонов приводит к целесообразности динамического (от поколения к поколению) увеличения или уменьшения весов операторов $w_j, j = \overline{1,n}$, в зависимости от количества особей, прошедших ранее (в предшествующих поколениях) по этим операторам. Динамическое изменение весов операторов можно представить в виде

$$\widetilde{w}_j^{(q)} = Ph_j^{(q)} w_j,\ j = \overline{1,n};\ q = \overline{1,Q}, \qquad (3.9)$$

где $\widetilde{w}_j^{(q)}$ – вес, присваиваемый оператору j в поколении q, $Ph_j^{(q)}$ – мультипликатор веса операторов j в поколении q $(0 \le Ph_j^{(q)} \le 1)$, Q – число поколений (итераций ГА). С учетом зависимости (3.9) динамический вариант компоненты F_1 функции приспособленности будет иметь вид:

$$F_1^{(q)}(x_i) = \sum_{j=1}^{n} \widetilde{w}_j^{(q)} g_j(x_i) = \sum_{j=1}^{n} Ph_j^{(q)} w_j g_j(x_i);\ q = \overline{1,Q}. \qquad (3.10)$$

В выражении (3.10) очень важно определить зависимость мультипликатора $Ph_j^{(q)}$ $(0 \le Ph_j^{(q)} \le 1)$ от аргументов, с тем чтобы веса операторов в функции приспособленности своевременно реагировали на покрытие операторов в предшествующих поколениях. От выбора способа варьирования $Ph_j^{(q)}$ зависит полученное в итоге разнообразие популяции наборов тестовых данных, а, следовательно, и степень покрытия ими тестируемого кода.

Для исследования выбраны две основные стратегии начального поведения мультипликатора $Ph_j^{(q)}$ в зависимости от номера поколения q – прямая и обратная стратегии. При прямой стратегии в первом поколении полагаем $Ph_j^{(1)} = 0$, и далее это значение увеличивается (или остается прежним) в зависимости от покрытия (или непокрытия) оператора j в предыдущем поколении. При обратной стратегии, наоборот, в первом поколении полагаем $Ph_j^{(1)} = 1$, и далее это значение

уменьшается (или остается прежним) в зависимости от покрытия (или непокрытия) оператора j в предыдущем поколении.

При обеих стратегиях возможно достижение мультипликатором границ интервала [0, 1]. Так, при прямой стратегии значение $Ph_j^{(q)}$ монотонно увеличивается, но по достижении предельного значения $Ph_j^{(q)} = 1$ (это значение соответствует максимальному приоритету оператора j в функции приспособленности) необходимо начать его уменьшение, чтобы обратить внимание эволюции на другие, еще непокрытые операторы тестируемого кода. Далее, после достижения минимально возможного значения $Ph_j^{(q)} = 0$, соответствующего невключению оператора j в функцию приспособленности, снова начинаем монотонное увеличение, и т.д. При обратной стратегии изменения происходят в противоположных направлениях, сначала в сторону уменьшения, затем в сторону увеличения, и т.д.

Такое колебательное изменение мультипликатора $Ph_j^{(q)}$ между значениями 0 и 1 может происходить с разной скоростью, задаваемой параметром ΔPh , что сказывается на общем числе колебаний внутри интервала [0,1] за время работы алгоритма. Обозначим $Trans_j^{(q)}$ – число полных проходов от 0 до 1 или от 1 до 0 мультипликатором $Ph_j^{(q)}$, совершенное к текущему поколению q. Тогда поведение мультипликатора для прямой стратегии может быть записано в виде

$$Ph_j^{(q)} = \begin{cases} 0, \text{ если } q = 1, \\ Ph_j^{(q-1)} + \Delta Ph \cdot (-1)^{Trans_j^{(q)}}, \text{ если } \widetilde{m}_j^{(q-1)} \neq 0, \\ Ph_j^{(q-1)}, \text{ если } \widetilde{m}_j^{(q-1)} = 0, \end{cases} \qquad (3.11)$$

а поведение мультипликатора для обратной стратегии – в виде

$$Ph_j^{(q)} = \begin{cases} 1, \text{ если } q = 1, \\ Ph_j^{(q-1)} - \Delta Ph \cdot (-1)^{Trans_j^{(q)}}, \text{ если } \widetilde{m}_j^{(q-1)} \neq 0, \\ Ph_j^{(q-1)}, \text{ если } \widetilde{m}_j^{(q-1)} = 0, \end{cases} \quad (3.12)$$

где $\widetilde{m}_j^{(q-1)}$– число хромосом в популяции, состоящей из m особей, покрывших оператор j в поколении $(q-1)$.

В работе предлагается несколько методов определения параметра скорости ΔPh. Метод *Half* предполагает, что один полный проход мультипликатора (от 0 до 1 или от 1 до 0) со скоростью ΔPh может быть получен при покрытии оператора j в половине поколений от исходно заданного числа поколений Q $(Q/2)$, метод *Quarter* – в четверти поколений $(Q/4)$, *Tenth* – в одной десятой от всех поколений $(Q/10)$. Чем меньше поколений (итераций) необходимо для одного полного прохода, тем больше будет параметр скорости ΔPh и тем чаще будут происходить колебания мультипликатора между предельными значениями [0, 1]. В таблице 3.4 представлены основные показатели, используемые для реализации предложенных методов варьирования мультипликатора $Ph_j^{(q)}$ с использованием постоянной скорости его изменения ΔPh. Методы с применением прямой стратегии помечены знаком плюс (+), а методы с обратной стратегией – знаком минус (-).

Таблица 3.4 – Методы определения параметра скорости ΔPh изменения мультипликатора $Ph_j^{(q)}$

Название метода	**Формула вычисления мультипликатора $Ph_j^{(q)}$**	**ΔPh**
Half+	(3.11)	$2/Q$
Half-	(3.12)	$2/Q$
Quarter+	(3.11)	$4/Q$
Quarter-	(3.12)	$4/Q$
Tenth+	(3.11)	$10/Q$
Tenth-	(3.12)	$10/Q$

Другой метод определения мультипликатора $Ph_j^{(q)}$, который мы назвали, *Count-* (метод основан на обратной стратегии), предполагает изменение его не на постоянную величину, а на величину, зависящую от интенсивности покрытия оператора j в предыдущем поколении:

$$Ph_j^{(q)} = \begin{cases} 1, \text{ если } q = 1, \\ Ph_j^{(q-1)}\left(1 - \widetilde{m}_j^{(q-1)}/m\right), \text{ если } \widetilde{m}_j^{(q-1)} \neq 0, \\ 1, \text{ если } \widetilde{m}_j^{(q-1)} = 0, \end{cases} \qquad (3.13)$$

где $\widetilde{m}_j^{(q-1)}$– число хромосом в популяции, состоящей из m особей, покрывших оператор j в поколении $(q-1)$.

В отличии от предложенных ранее методов, в *Count-* значение $Ph_j^{(q)}$ будет уменьшаться тем сильнее, тем больше хромосом в предыдущем поколении покрыли оператор j. Постепенного увеличения мультипликатора в данном случае не происходит, вместо этого, если оператор не был покрыт ($\widetilde{m}_j^{(q-1)} = 0$), то для него скачкообразно устанавливается максимальное значение $Ph_j^{(q)} = 1$. Таким образом, часто покрываемые операторы перестают играть значимой роли в процессе поиска тестовых наборов, и алгоритм будет в большей мере пытаться сформировать наборы для ещё не покрытых путей.

Проведём сравнение применения различных методов определения мультипликатора $Ph_j^{(q)}$, используя предложенные выше методы для тестовой программы SUT2. Рисунок 3.6 показывает сравнение среднего покрытия для различных значений параметра соотношения k компонент функции приспособленности (3.8), в которой F_1 вычисляется или по формуле (3.1), то есть без модификации, или по формуле (3.10), то есть при использовании модификации методами *Half+*, *Quarter+* и *Tenth+* и *Count-*. Среднее покрытие вычисляется на основе 1500 запусков, в качестве параметров ГА выбраны $Q = 50$, $m = 25$, при которых сравнительно редко достигается полное покрытие.

На рисунке 3.6 красным цветом выделено среднее покрытие при использовании формулы (3.1) (статический метод), оттенками синего – методы монотонного изменения Ph на основе прямой стратегии, чёрным цветом – метод *Count-*. Методы определения параметра ΔPh на основе обратной стратегии не представлены на рисунке, но, в целом, имеют приблизительно схожие значения среднего покрытия.

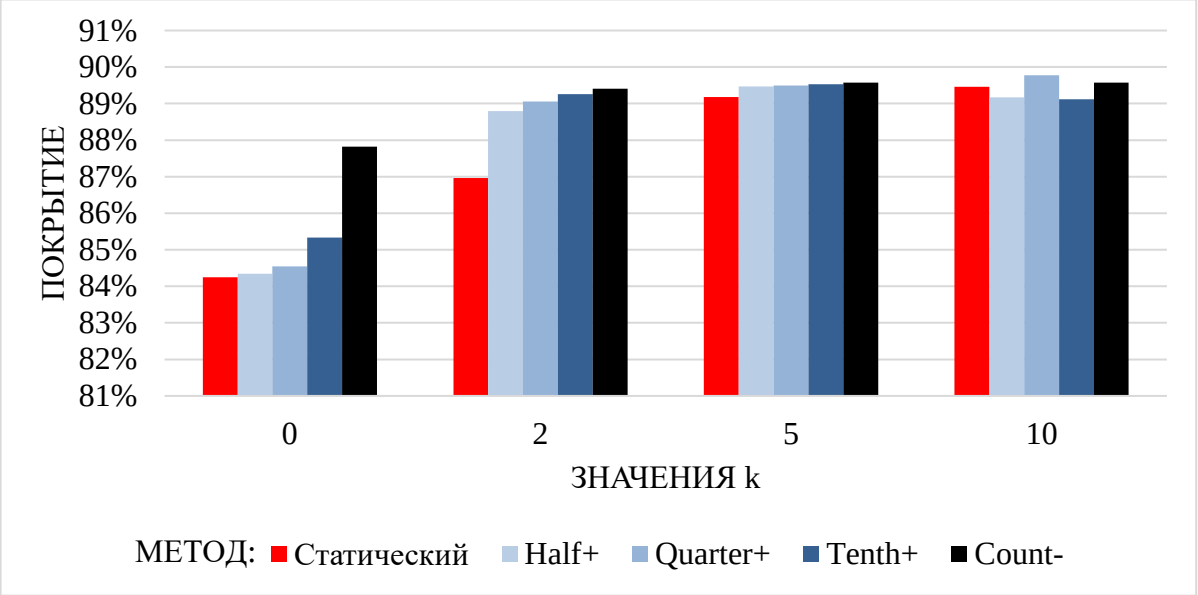
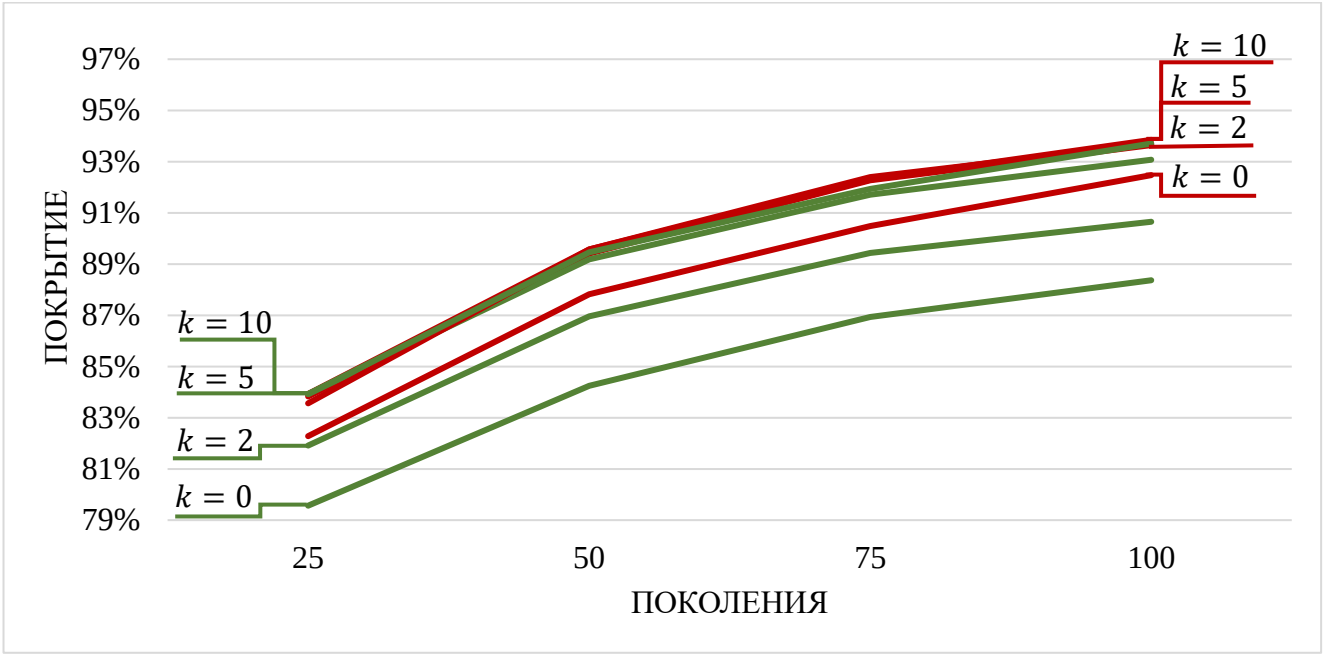


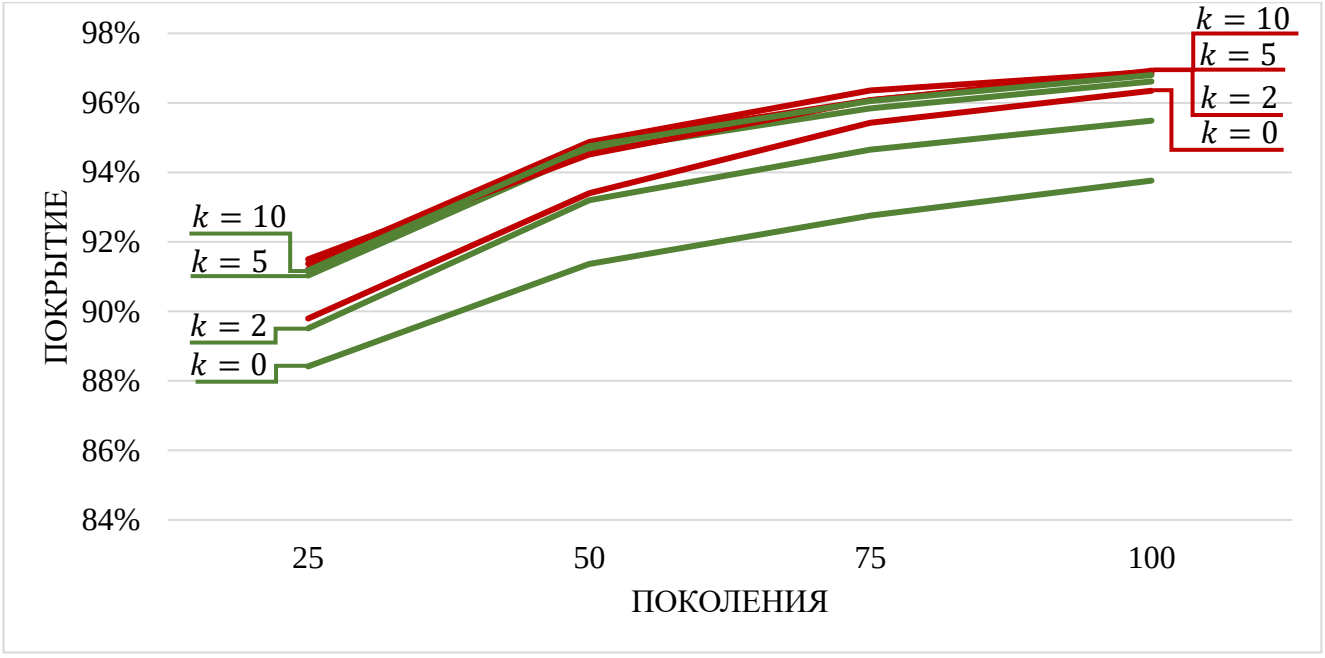
Рисунок 3.6 – Сравнение различных модификаций ($Q = 50, m = 25$)

Каждый из предложенных методов определения мультипликатора $Ph_j^{(q)}$ показал большее среднее значение покрытия, чем статический метод (без модификации), для каждого из рассматриваемых значений k. На рисунке 3.6 видно, что для статического метода среднее покрытие постепенно увеличивается с ростом k, тогда как при использовании различных методов модификации максимум среднего покрытия достигается уже при $k = 2$, и в дальнейшем не уменьшается. Наилучший результат среди всех рассмотренных методов показал *Count-*, поэтому именно данный метод будет использоваться при проведении дальнейших исследований данной модификации функции приспособленности.

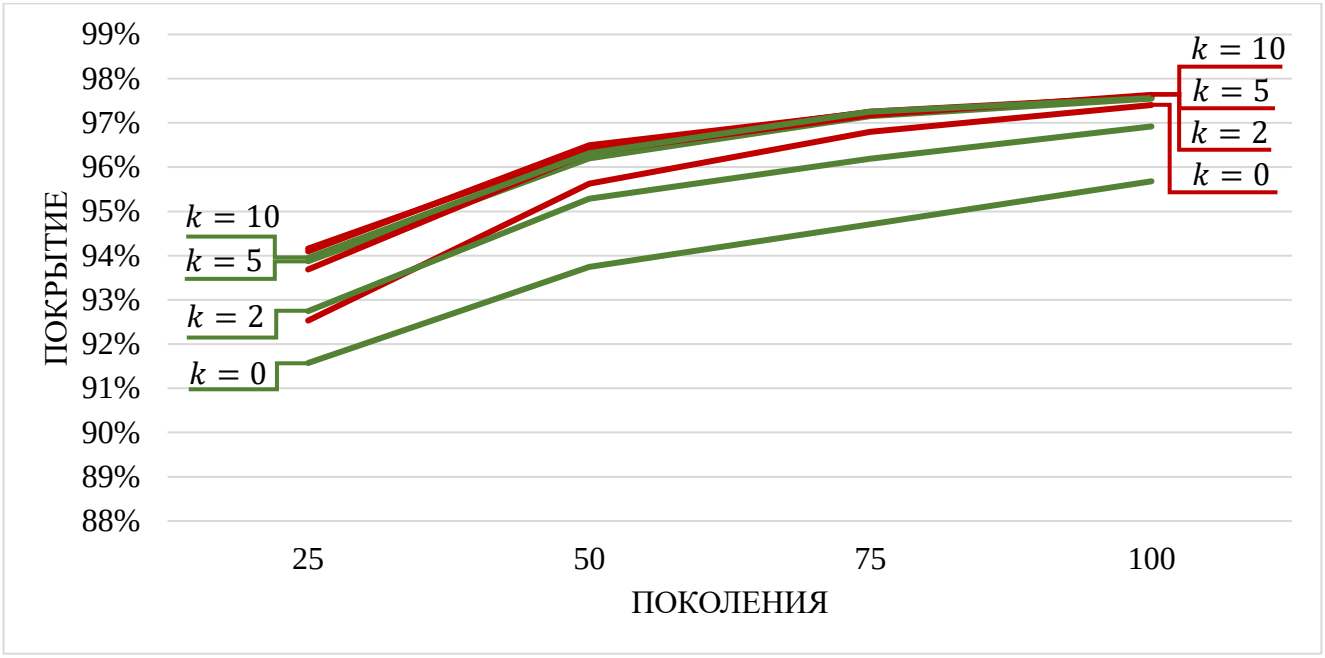
Рассмотрим, как алгоритм работает при различных значениях k для разного количества поколений Q и размера популяции m. На рисунке 3.7 значение среднего покрытия при указанных k без модификации показано зелёными линиями, значения k расположены слева, а с использованием модификации – красными, значения k – справа.



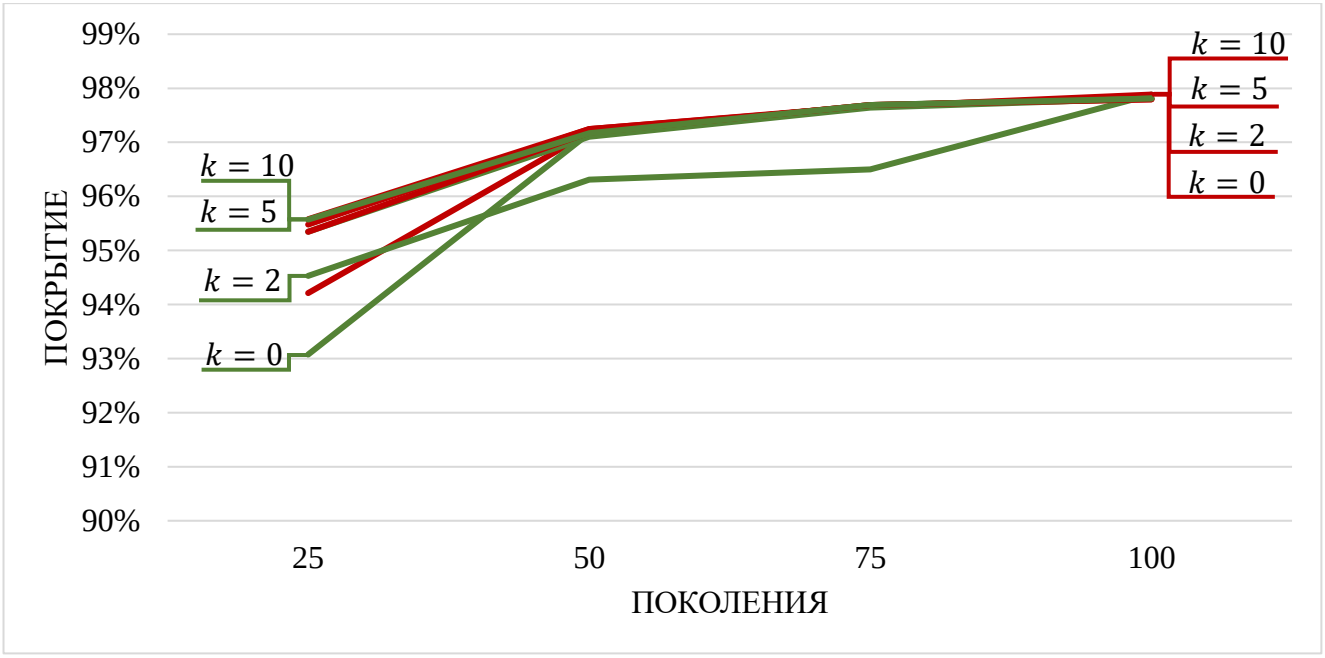
(a) Размер популяции $m = 25$



(b) Размер популяции $m = 50$



(c) Размер популяции $m = 75$



(d) Размер популяции $m = 100$

Рисунок 3.7 – Сравнение покрытия в срезе размера популяции и количества поколений

Анализ результатов, представленных на рисунке 3.7, позволяет сделать вывод, что модификация позволяет существенно увеличить покрытие даже без использования определённого ранее оптимального значения $k = 10$, что отчётливо

заметно при малых размерах популяции. При этом, даже при $k = 0$, то есть без использования аддитивной компоненты F_2 функции приспособленности, достигается более высокое покрытие, чем без модификации.

Более показательно сравнение среднего покрытия без использования модификации и с использованием самого лучшего метода модификации Count- можно увидеть на рисунке 3.8. На нём показано среднее покрытия при параметрах алгоритма $Q = 50, m = 25$.

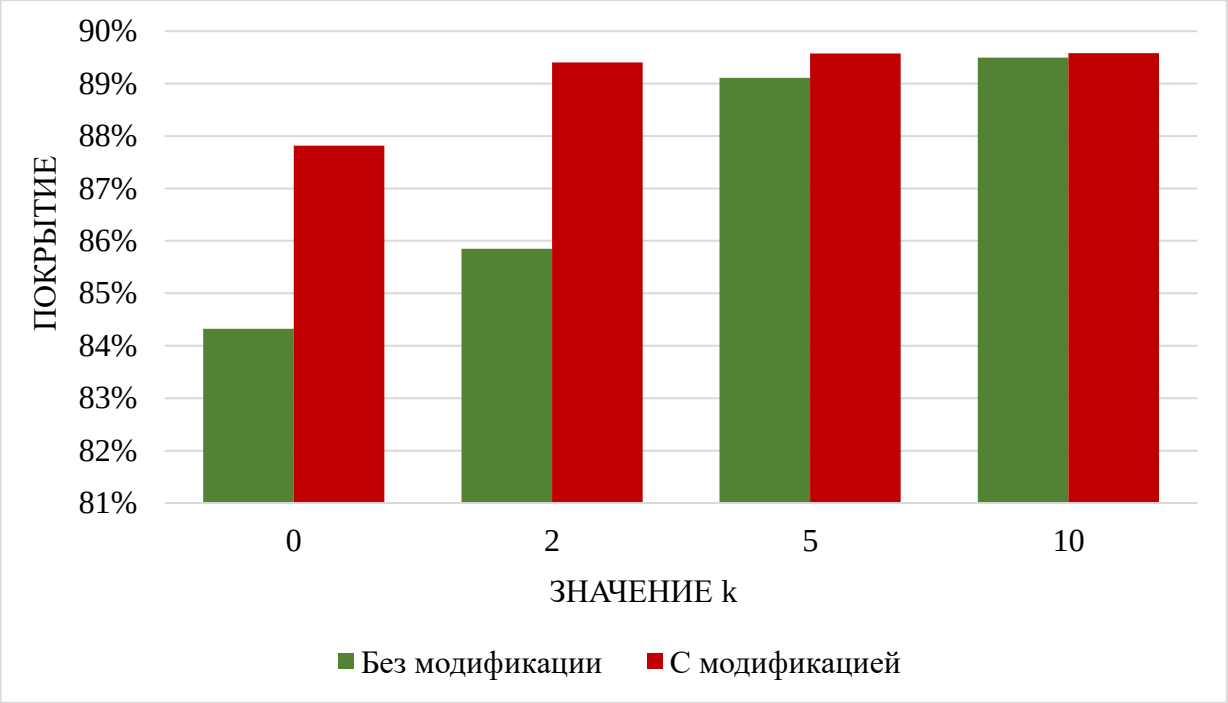


Рисунок 3.8 – Сравнение покрытия с использованием модификации и без неё ($Q = 50, m = 25$)

Таким образом, использование модификации позволяет увеличить среднее покрытие, которое особенно заметно при $k = 0$. Что более важно, максимальное покрытие достигается при использовании любого отличного от нуля значения k, то есть параметр соотношения компонент функции приспособленности k перестаёт играть значимой роли для достижения максимального покрытия. В результате, предложенная модификация на основе динамического изменения весов операторов

позволяет увеличить покрытие кода при генерации тестовых наборов, а также устранить необходимость определять значение k для каждой отдельной тестируемой программы.

Выводы по главе 3

В данной главе были получены следующие основные результаты:

1. Проведён анализ влияния параметров ГА на скорость его работы. Выявлена линейная зависимость между временем работы и количеством поколений и нелинейная – между временем работы и размером популяции. Проведённые исследования позволяют сделать вывод, что размер популяции оказывает большее влияние на быстродействие алгоритма, чем число поколений.

2. Описаны результаты исследования методов смешивания для увеличения разнообразия популяции. Результаты показали, что методы прямого и сдвигающегося смешивания позволяют достичь большего разнообразия.

3. Предложена модификация функции приспособленности введением дополнительной аддитивной компоненты F_2, отдающая приоритет тем наборам тестовых данных, которые инициируют прохождение по наиболее различающимся путям на графе потоков управления. Проведенные исследования показывают высокую степень покрытия, обеспечиваемую с использованием данной модификации.

4. Предложена модификация функции приспособленности с динамическим изменением весов операторов компоненты F_1 в зависимости от степени их покрытия в предшествующем поколении, что позволило увеличить покрытие кода и устранить необходимость априорного задания параметра k, устанавливающего соотношение между компонентами функции приспособленности.

ГЛАВА 4 РЕАЛИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ДЛЯ АНАЛИЗА ИСХОДНОГО КОДА И ПОЛУЧЕНИЯ ТЕСТОВЫХ НАБОРОВ

Данная глава посвящена программной реализации предлагаемого алгоритма генерации тестовых данных на основе анализа тестируемого кода. Так как алгоритм работает в динамической среде, тестируемый код будет скомпилирован встроенным компилятором, поэтому для тестовой программы был выбран высокоуровневый язык C#. Представленное приложение позволяет получать тестовые данные для последующего её тестирования. Приложение реализовано в интегрированной среде разработки Microsoft Visual Studio 2019 (VS) с использованием языка C# на базе фреймворка .NET Framework 4.7.2. Имеются два свидетельства о регистрации программы для ЭВМ [13, 14].

4.1 Описание и структура разработанного приложения

Приложение позволяет получать наборы тестовых данных многократным запуском метода генерации данных для одного пути и одновременным подбором для множества путей. Структура приложения определяется следующим образом (Рисунок 4.1):

– Тестируемый код. На вход разработанного приложения подается тестируемый код, для которого необходимо сгенерировать тестовые наборы x. Код в виде текста вводится в соответствующее поле интерфейса, что позволяет взаимодействовать с ним в процессе работы приложения.

– Интерфейс. Обеспечивает взаимодействие между приложением и пользователем. В модуле интерфейса производится передача тестируемого кода в реализацию, и обеспечивается вывод выходной информации.

– Реализация. В модуле реализации производится обработка и анализ тестируемого кода и генерация наборов тестовых данных.

– Сгенерированные данные. Выходной информацией для приложения являются сгенерированные тестовые наборы x. В качестве дополнительной

информации выводится значение индикатора покрытия $g(x)$. При необходимости можно также обеспечить вывод всех операторов с указанием значения индикатора.

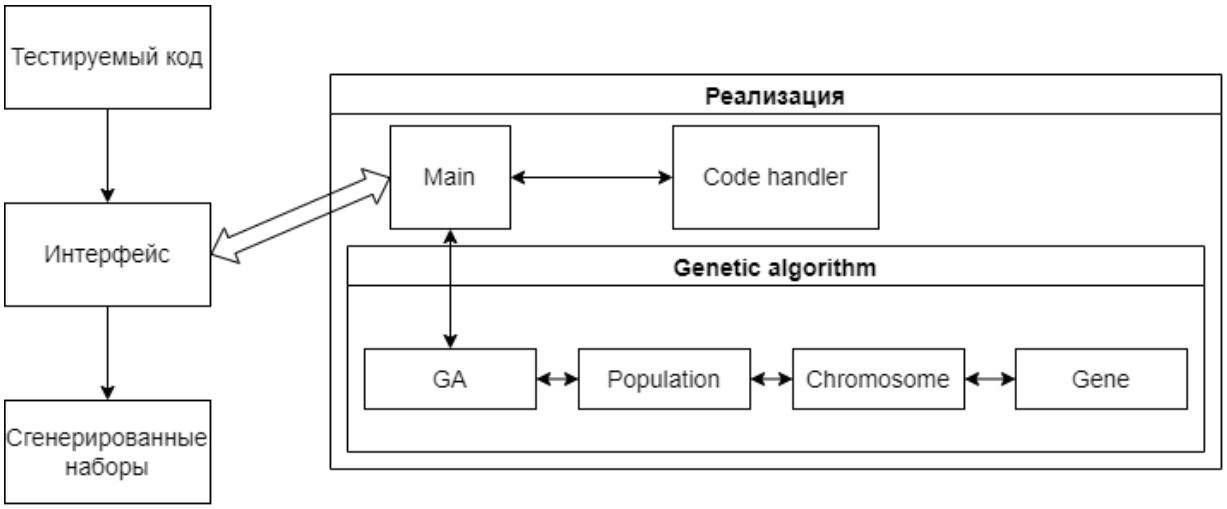


Рисунок 4.1 – Структурная схема разработанного приложения

Реализация разделена на следующие модули:

– Main. Основная часть приложения, в которой происходит получение данных из интерфейса, запуск функций обработки кода и генерации данных, вывод выходной информации. Формат вывода рассматривается в п. 4.2.1.

– Code handler. Модуль обработки кода, в котором производится анализ кода и определяются операторы, условия и циклы. Для каждого оператора вычисляется вес w и динамический мультипликатор Ph. В данном модуле код компилируется в сборку.

– Genetic algorithm. Модуль, в котором происходит запуск генетического алгоритма для генерации данных, содержит следующие классы:

  ○ GA. Основной класс ГА, реализующий запуск основных эволюционных операций (п. 2.2.3) из других классов и обеспечивающий получение нового поколения переносом лучших хромосом предыдущего поколения и скрещиванием.

- Population. Класс, реализующий понятие популяции. В нем определена функция заполнения популяции случайными хромосомами при инициировании, производится вычисление аддитивной компоненты приспособленности F_2 функции приспособленности и определяется значение покрытия каждого оператора и популяции в целом.
- Chromosome. Определяются основные параметры хромосомы, формируются новые хромосомы со случайными генами, производится вычисление компоненты функция приспособленности F_1, реализована функция скрещивания между двумя хромосомами.
- Gene. Низший уровень реализации ГА, в котором хранится значение гена, определены функции генерации случайного значения и эволюционной операции мутации.

Структура кода определена таким образом, чтобы отдельные функциональные части ГА, в которых производятся эволюционные операции, соответствовали его основным понятиям.

4.2 Графический интерфейс пользователя

Приложение реализовано в среде Visual Studio (VS) с использованием встроенной графической подсистемы Windows Presentation Foundation (WPF). Приложение содержит одно основное окно, в котором производится ввод и вывод данных и обеспечивается настройка и запуск алгоритма (Рисунок 4.2).

Исходный код тестируемой программы вводится в левое поле «Source code». Код представляется в виде обычного текста, который будет обработан и проанализирован для определения весов операторов, после чего скомпилирован. Результаты работы программы выводятся в поле «Output».

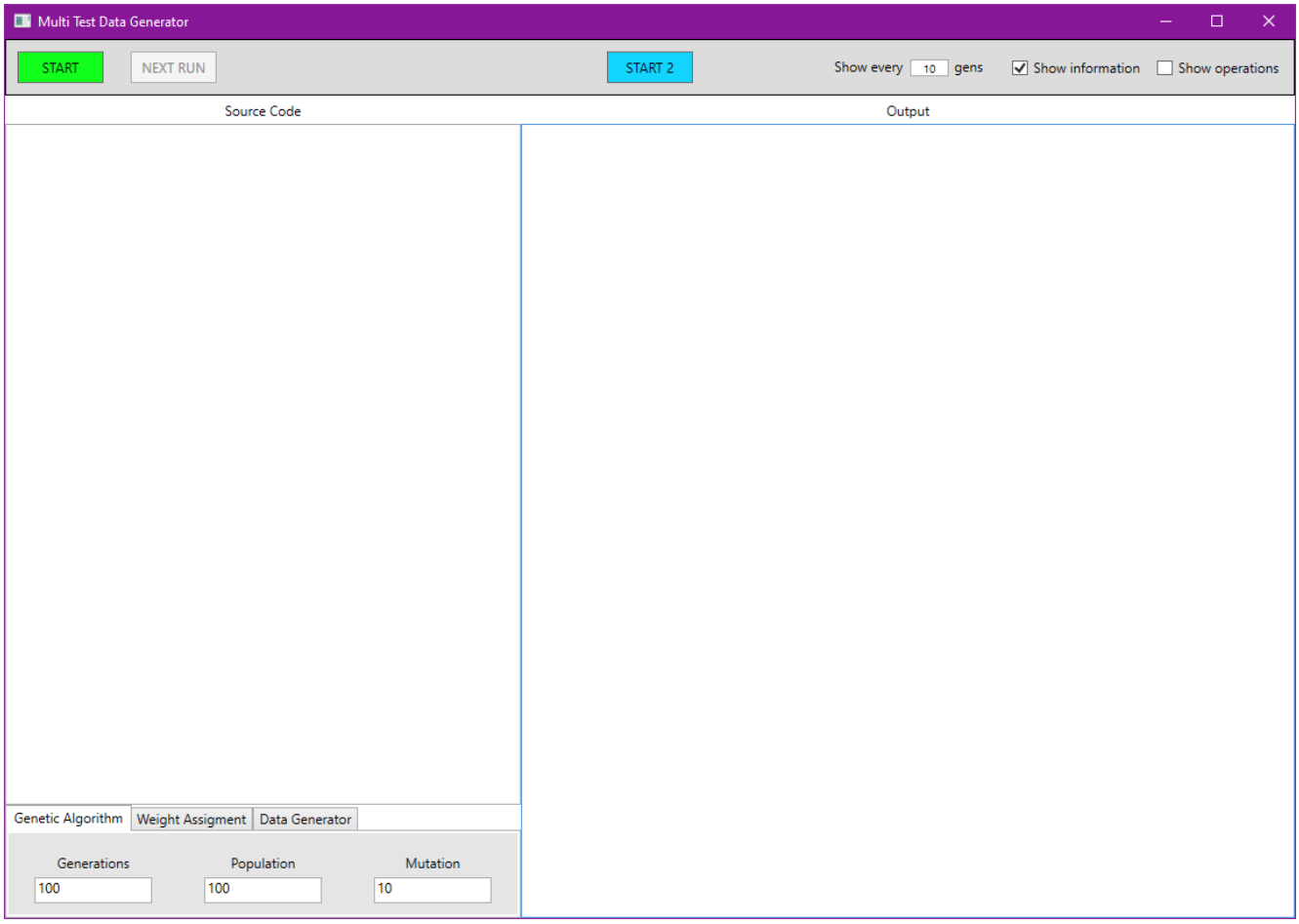


Рисунок 4.2 – Интерфейс приложения на момент запуска без введенных данных

Пример работы приложения с заполненными входными данными и результатами показан ниже (Рисунок 4.3). В качестве входных данных на рисунке используется анализируемая программа SUT2.

Так как тестируемый код подается в виде текста, то у него нет возможности взаимодействовать с другими файлами. Поэтому все функции и процедуры, которые используются в нем, должны быть реализованы в этом же тексте, либо во встроенных библиотеках C#. Тем не менее, возможно использовать динамические библиотеки с расширением .dll, если поместить их в папку с исполняемым файлом приложения.

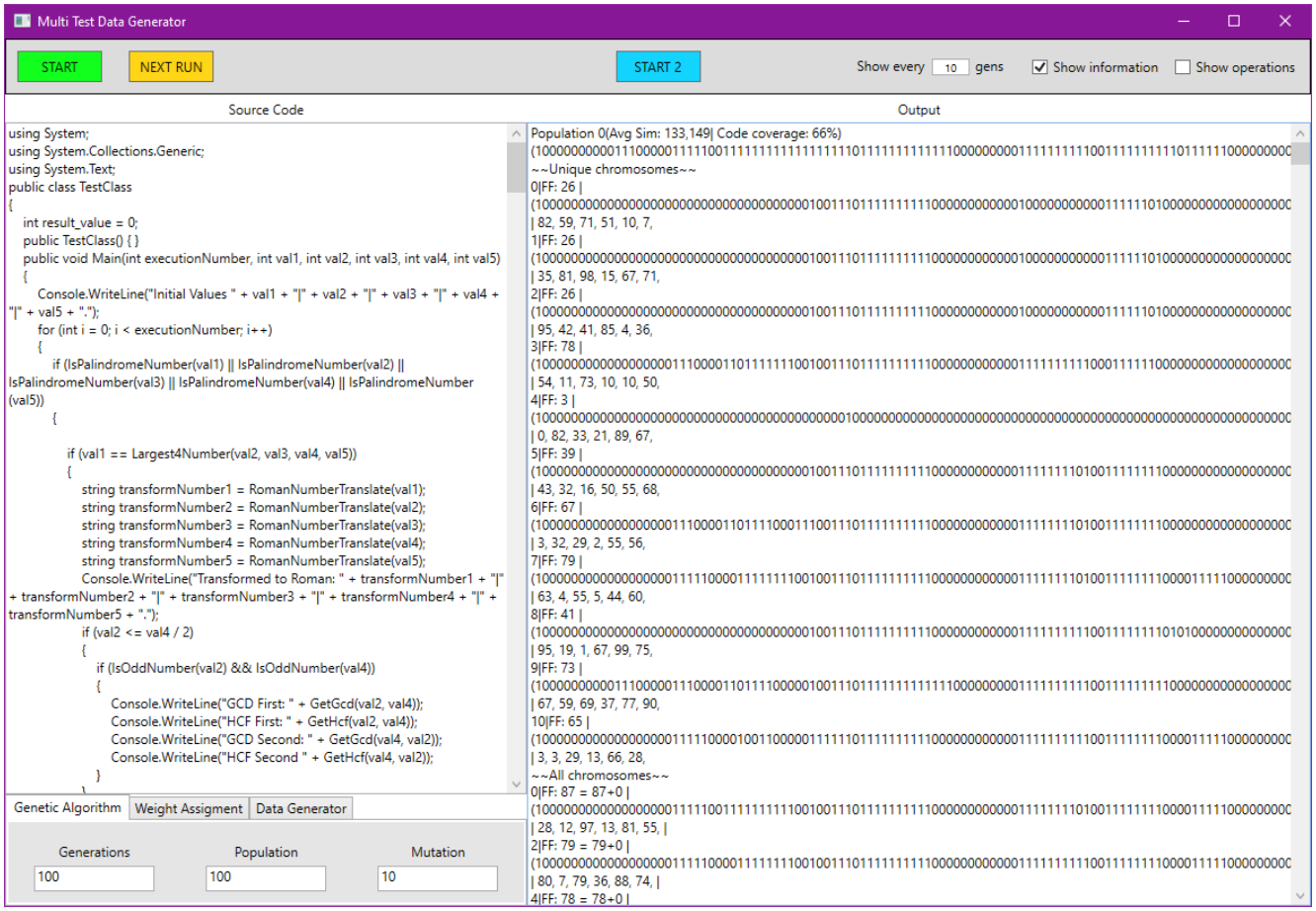


Рисунок 4.3 – Интерфейс с введенным исходным кодом в поле для ввода и результатами в поле для вывода

4.2.1 Поля для ввода исходного кода и вывода результатов

Основная часть интерфейса приходится на два текстовых поля, первое – для ввода тестируемого кода, а второе – для вывода результатов (Рисунок 4.4).

Поле слева под номером 1 является областью для ввода исходного кода тестируемой программы. Справа, под номером 2, находится поле для вывода, в которое выводятся результаты работы алгоритма генерации данных.

Кроме этого, поле 2 служит для вывода сообщения об ошибках при компиляции кода (Рисунок 4.5).

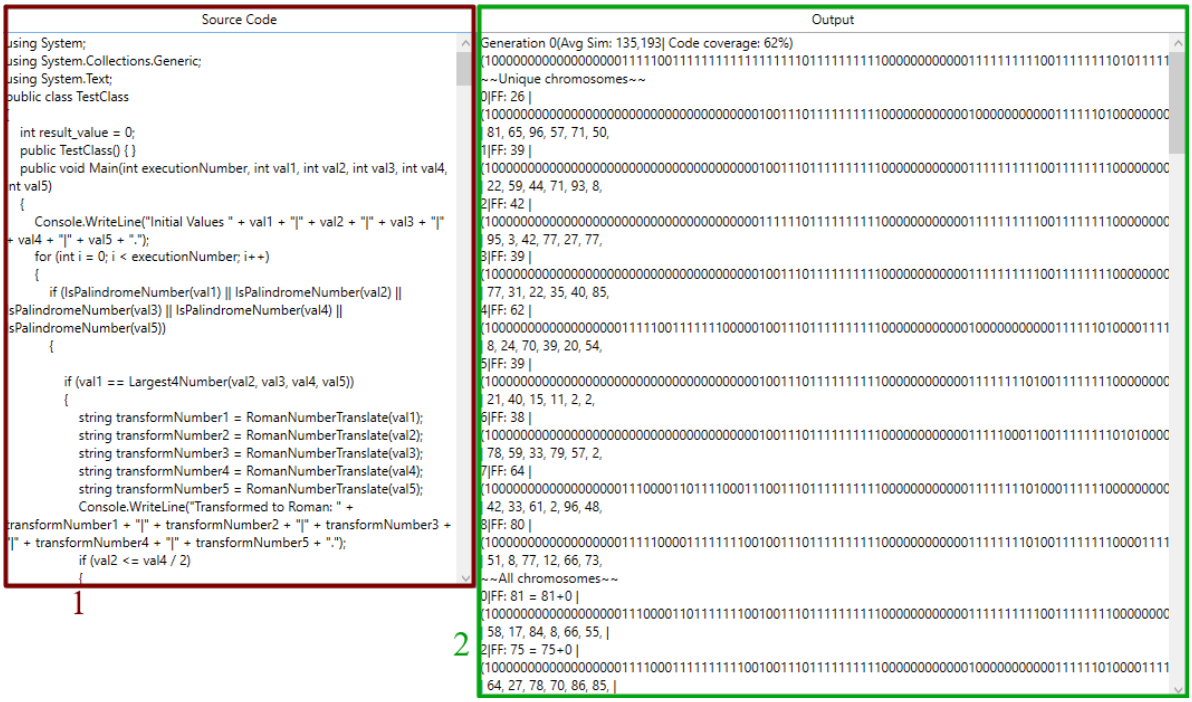


Рисунок 4.4 – Поля для ввода исходного кода (1) и для вывода (2)

```
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,845) : error CS0103: The name 'Largest4Numberas' does not exist in the current context
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,11659) : warning CS0162: Unreachable code detected
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,12450) : warning CS0162: Unreachable code detected
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,12696) : warning CS0162: Unreachable code detected
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,12872) : warning CS0162: Unreachable code detected
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,12996) : warning CS0162: Unreachable code detected
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,13183) : warning CS0162: Unreachable code detected
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,13893) : warning CS0162: Unreachable code detected
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,14018) : warning CS0162: Unreachable code detected
125c:\Users\Zores\AppData\Local\Temp\cgfgcoxg.0.cs(1,14215) : warning CS0162: Unreachable code detected
```

Рисунок 4.5 – Пример сообщения об ошибках при невозможности скомпилировать код из-за неверного названия функции

На рисунке 4.5 показано сообщение, выдаваемое при обращении к несуществующей функции. Если такие ошибки выдаются, то их необходимо исправить до запуска алгоритма генерации тестовых данных.

Если код был успешно скомпилирован, то запускается алгоритм генерации данным производится вывод выходной информации. Вывод представляется в зависимости от выбранного метода генерации данных, то есть метода генерации

данных для одного пути (Рисунок 4.6) и метода генерации для множества путей (Рисунок 4.7).

```
Generation 100(Code coverage: 52%)
0|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 63, 16, 71, |
2|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 92, 75, 94, |
4|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 74, 78, 99, |
6|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 69, 60, 86, |
8|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 77, 67, 80, |
10|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 72, 14, 76, |
12|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 68, 17, 89, |
14|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 76, 17, 94, |
16|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 64, 37, 79, |
18|FF: 164100 | (11111111111000000000000000001111111111111111110000000001) | 72, 45, 95, |
```

Рисунок 4.6 – Вывод при запуске генерации данных для одного пути

```
Generation 100(Avg Sim: 37,2896| Code coverage: 100%)
(11111111111111111111111111111111111111111111111111111111)
~~Unique chromosomes~~
0|FF: 15 | (11111111111111000000000000000000000000000000000000000001) | 45, 12, 81,
1|FF: 29 | (11111111111000000000000000001111111111111111110000000001) | 65, 25, 74,
2|FF: 35 | (11111111111000000000001111111111111000000001111111111) | 1, 14, 73,
3|FF: 20 | (11111111111000111111110000000000000000000000000000000001) | 60, 5, 78,
~~All chromosomes~~
0|FF: 62,996 = 20+42,996 | (11111111111000111111110000000000000000000000000000000001) | 60, 5, 78, |
2|FF: 45,204 = 15+30,204 | (11111111111111000000000000000000000000000000000000000001) | 11, 40, 74, |
4|FF: 41,198 = 29+12,198 | (11111111111000000000000000001111111111111111110000000001) | 66, 21, 72, |
6|FF: 37,132 = 29+8,132 | (11111111111000000000000000001111111111111111110000000001) | 61, 59, 73, |
8|FF: 35,868 = 35+0,868 | (11111111111000000000001111111111111100000000011111111111) | 3, 21, 99, |
10|FF: 35,521 = 35+0,521 | (11111111111000000000001111111111111100000000011111111111) | 2, 44, 100, |
```

Рисунок 4.7 – Вывод при использовании метода генерации данных для множества путей

В первой строке выводится номер поколения и значение покрытия. Для метода генерации данных для множества путей дополнительно выводится значение среднего сходства в популяции, а ниже – список хромосом в следующем формате:

Номер*|*Функция приспособленности*|*Покрытые операторы*|*Набор

Обозначения расшифровываются следующим образом:

Номер	порядковый номер хромосомы.
Функция приспособленности	значение функции приспособленности для хромосомы.
Набор	сгенерированный тестовый набор.

Покрытые операторы	операторы, которые покрываются данным тестовым набором, вектор $g(x)$: 1 – тестовый набор покрывает оператор; 0 – тестовый набор не покрывает.

Для метода генерации одного тестового набора решением является лучшая хромосома последнего поколения. Для метода генерации данных множества тестовых наборов решением является пул элитных хромосом.

4.2.2 Обработка команд запуска алгоритмов и настройка вывода

В верхней части интерфейса находятся кнопки для запуска алгоритма генерации тестовых данных (Рисунок 4.8).

Рисунок 4.8 – Кнопки для запуска алгоритма

Кнопка «START» запускает алгоритм генерации данных для одного пути кода (п. 2.5.3). Данный метод генерирует тестовый набор для одного самого сложного пути. Кнопка «START 2» инициирует работу алгоритма генерации данных для множества путей.

Кнопка «NEXT RUN» позволяет продолжить поиск дополнительных наборов данных, если при нажатии кнопок «START» или «START 2» не было достигнутого желаемого покрытия. Для ранее покрытых операторов происходит обнуление весов, поэтому алгоритм будет генерировать тестовые наборы для самого сложного пути из ещё непокрытых. Таким образом, применение «NEXT RUN» позволяет реализовать функционал постепенного увеличения покрытия многократным запуском.

Справа вверху находятся элементы графического интерфейса, которые позволяют настроить вывод информации (Рисунок 4.9). Первое поле определяет количество итераций ГА, через которые сгенерированные наборы данных выводятся на экран, что может быть использовано для лучшего отслеживания процесса генерации данных.

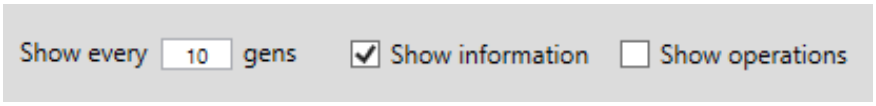


Рисунок 4.9 – Элементы интерфейса для настройки вывода

Флаг «Show information» используется для включения и выключения расширенного вывода данных. При отключенной опции вывод ограничивается только средним значением функции приспособленности и значением покрытия популяцией (Рисунок 4.10).

Generation 0(Avg Sim: 130,7282| Code coverage: 84%)
Generation 5(Avg Sim: 125,0494| Code coverage: 91%)
Generation 10(Avg Sim: 126,0236| Code coverage: 96%)
Generation 15(Avg Sim: 124,1358| Code coverage: 96%)
Generation 20(Avg Sim: 122,7716| Code coverage: 96%)
Generation 25(Avg Sim: 125,781| Code coverage: 98%)
Generation 30(Avg Sim: 124,5788| Code coverage: 98%)
Generation 35(Avg Sim: 124,1908| Code coverage: 98%)
Generation 40(Avg Sim: 123,4364| Code coverage: 98%)
Generation 45(Avg Sim: 124,6948| Code coverage: 98%)
Generation 50(Avg Sim: 124,673| Code coverage: 98%)
Generation 55(Avg Sim: 124,9002| Code coverage: 98%)
Generation 60(Avg Sim: 124,7594| Code coverage: 98%)
Generation 65(Avg Sim: 123,8864| Code coverage: 98%)
Generation 70(Avg Sim: 125,882| Code coverage: 98%)
Generation 75(Avg Sim: 124,0522| Code coverage: 98%)
Generation 80(Avg Sim: 126,961| Code coverage: 98%)
Generation 85(Avg Sim: 125,6652| Code coverage: 98%)
Generation 90(Avg Sim: 125,146| Code coverage: 98%)
Generation 95(Avg Sim: 122,4884| Code coverage: 98%)
Generation 100(Avg Sim: 121,835| Code coverage: 98%)

Рисунок 4.10 – Вывод каждых 5 поколений с отключенной опцией Show Information

Флаг «Show operations» используется для вывода всех операторов кода с соответствующим значением вектора индикатора покрытия $g(x)$. Данная функция

необходима для определения операторов, которые не были покрыты в результате работы алгоритма, что так как позволяет локализовать потенциально недостижимые операторы при последующей верификации.

На рисунке 4.11 показано соотношение тестируемого кода и вывода со включенной опцией Show operations. В качестве кода выступает тестовая программа функции целочисленного деления. Все операторы в функции были покрыты за исключением $res = (int) - result$ и $res = int.MaxValue$. Эти операторы не были покрыты, так как предшествующие им условия не могли быть достигнуты при текущих ограничениях входных переменных.

```
public int Divide(int dividend, int divisor)
   {
      uint x = dividend > 0 ? (uint)dividend : (uint)-dividend;
      uint y = divisor > 0 ? (uint)divisor : (uint)-divisor;
      uint result = 0, z = 0;
      var idx = 0;
      while (x >= y)
      {
         z = y;
         for (idx = 0; x >= z && z != 0; idx++, z *= 2)
         {
            x -= z;
            result += (uint)1 << idx;
         }
      }
      int res = 0;
      if ((dividend ^ divisor) >> 31 == -1)
      {
         res = (int)-result;
      }
      else
      {
         if (result > int.MaxValue)
         {
            res = int.MaxValue;
         }
         else
         {
            res = (int)result;
         }
      }
      return res;
   }
```

```
1: uint x = dividend > 0 ? (uint)dividend : (uint)-dividend
1: uint y = divisor > 0 ? (uint)divisor : (uint)-divisor
1: uint result = 0, z = 0
1: var idx = 0


1: z = y


1: x -= z
1: result += (uint)1 << idx


1: int res = 0


0: res = (int)-result




0: res = int.MaxValue




1: res = (int)result


1: return res
```

Рисунок 4.11 – Результаты работы алгоритма. Слева представлен тестируемый код, справа – операторы кода с соответствующими индикаторами покрытия.

4.2.3 Настройка параметров генетического алгоритма

В нижней левой части интерфейса находится область, в которой определяются основные параметры ГА (Рисунок 4.12). Количество поколений и размер популяции задаются в зависимости от желаемой степени покрытия. В поле Generation вносится количество поколений, в поле Population – размер популяции, а в поле Mutation – шанс мутации одного гена (в процентах).

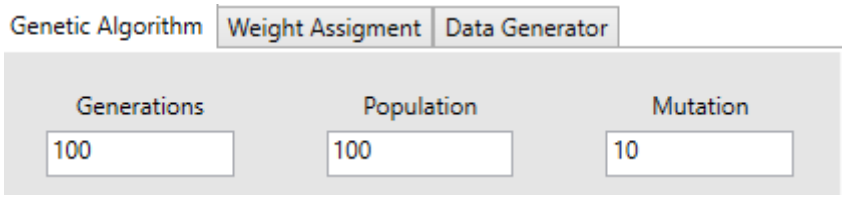


Рисунок 4.12 – Часть интерфейса, ответственная за настройку алгоритма. Вкладка с параметрами ГА.

Во второй вкладке (Рисунок 4.13) происходит выбор необходимой метрики оценки сложности кода (NOD, SLOC, ABC или Jilb) и устанавливается значение k для определения соотношения между компонентами функции приспособленности F_1 и F_2.

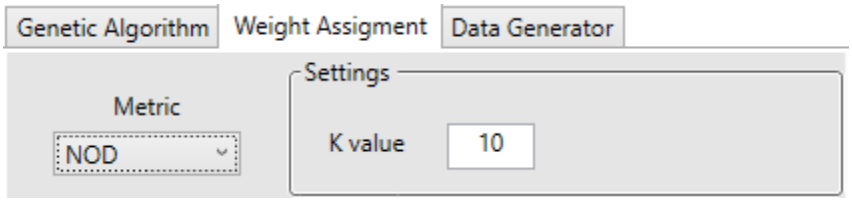


Рисунок 4.13 – Вкладка с параметрами назначения весов

Последняя вкладка (Рисунок 4.14) необходима для установки минимального (поле Min value) и максимального (поле Max value) значений входных переменных, за которые они не должны выходить.

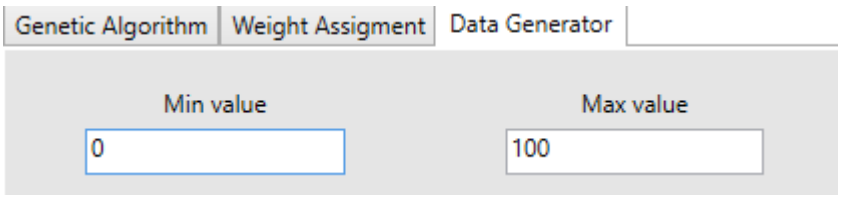


Рисунок 4.14 – Вкладка, в которой определяются минимальное и максимальное значения переменных

4.3 Реализация модулей работы алгоритма

При нажатии на одну из кнопок запуска алгоритма, происходит считывание тестируемого кода, который передается в модуль обработки кода (Code handler).

4.3.1 Модуль обработки тестируемого кода

В данном модуле происходит анализ тестируемого кода, вычисление значений весов w и динамического мультипликатора весов Ph, компиляция кода в сборку. Из-за особенностей объектно-ориентированный среды, генерация тестовых наборов происходит для функции *«Main»*.

Текст тестируемого кода обрабатывается построчно. Если в строке содержится определение новой функции, то последующие вложенные операторы определяются как внутренние. Если встречается цикл или условие, то для вложенных операторов уровень вложенности увеличивается на единицу (используется при вычислении метрики NOD).

Далее проверяется, является ли рассматриваемая строка оператором назначения или запуском функции. Если является, то данная строка рассматривается в качестве оператора, ей назначается вес и определяется начальное значение мультипликатора Ph. В дальнейшем, при каждой итерации ГА, будет происходить обращение к функции перерасчёта мультипликатора Ph, зависящего от покрытия оператора в предыдущем поколении.

После завершения анализа тестируемой программы и определения весов *w* для каждого оператора, код компилируется в сборку. Если в коде не содержится ошибок, производится запуск генетического алгоритма.

4.3.2 Модуль генерации данных генетическим алгоритмом

Первым этапом работы ГА является формирование начальной популяции. Первая популяция формируется случайным образом. После формирования случайной популяции производится последовательный запуск модулей, реализующих эволюционные операции ГА, в соответствии с описанием на рисунке 2.2. Классовая диаграмма модуля представлена на рисунке 4.15.

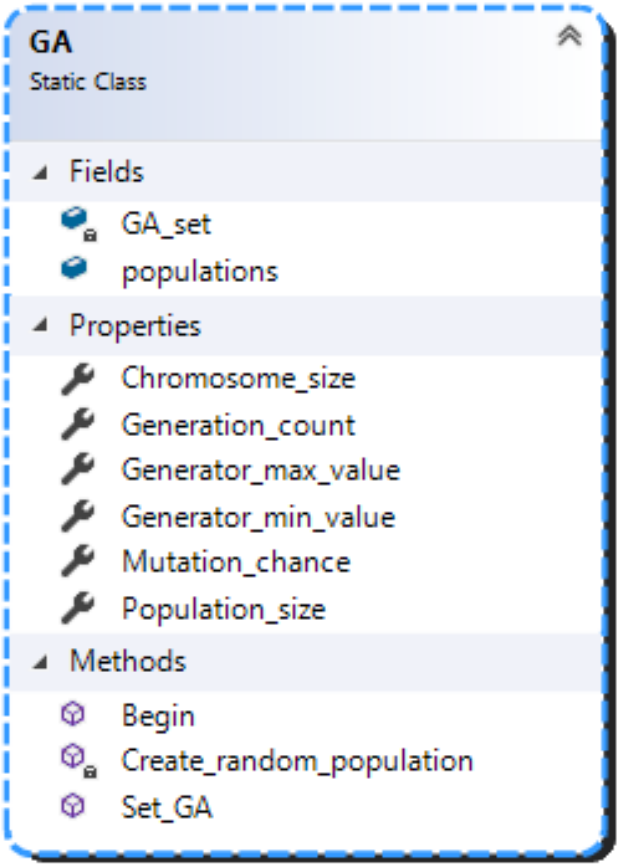


Рисунок 4.15 – Диаграмма статического класса GA с полями и функциями

Общая структура модуля GA

Общая структура всего модуля показана на рисунке 4.16. С помощью линий показывается отношения между классами.

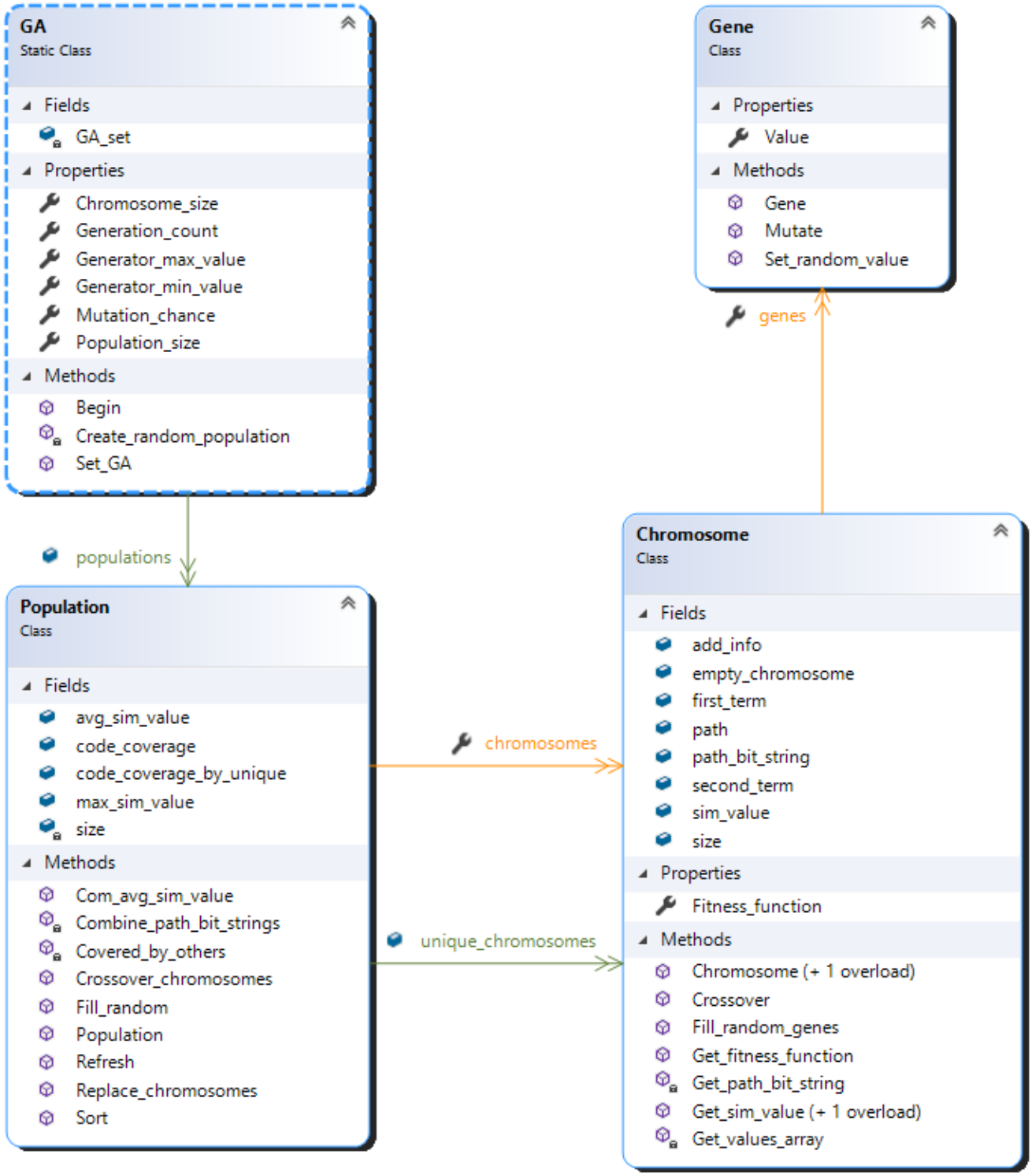


Рисунок 4.16 – Общая структура модуля реализации генетического алгоритма

Класс «Популяция»

Класс «Популяция» содержит весь список хромосом в текущем поколении. Данный класс обеспечивает реализацию эволюционных операций, применяемых к популяции в целом, а именно, вычисление функции приспособленности и операцию скрещивание (Рисунок 4.17).

Также данный класс реализует вычисление аддитивной компоненты F_2 функции приспособленности, так как для её определения необходимо использовать значение схожести всех хромосом в популяции.

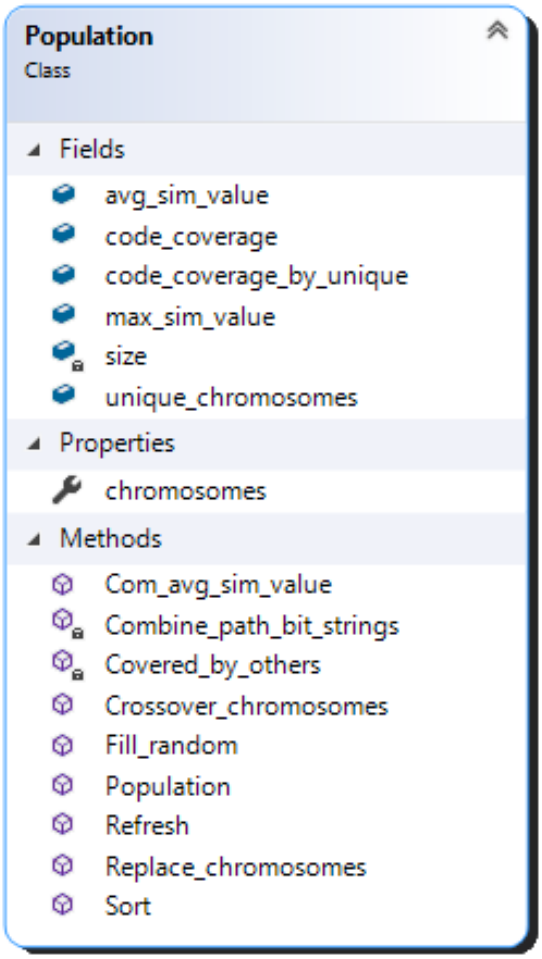


Рисунок 4.17 – Диаграмма класса Population, определяющая реализацию популяции

Класс «Хромосома»

Следующим классом в структуре модуля ГА является класс «Хромосома» (Рисунок 4.18), котором хранится список генов, значение функции приспособленности для каждой из хромосом, путь, по которому она проходит, и вычисляется компонента F_1. Также в данном классе путь, инициируемый тестовым набором, преобразуется из списка операторов в вид битовой строки $g(x)$.

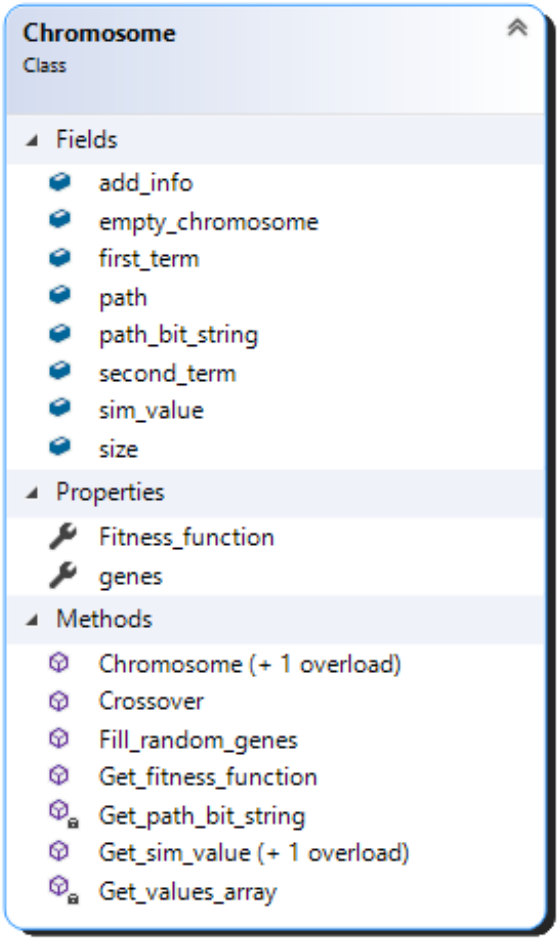


Рисунок 4.18 – Диаграмма класса Chromosome, показывающая реализацию хромосом

Класс «Ген»

Низшим классом в структуре генетического алгоритма является класс «Ген» (Рисунок 4.19). Его структура максимально проста – в нем хранится значение гена и реализованы функция генерации случайной величины и операция мутации.

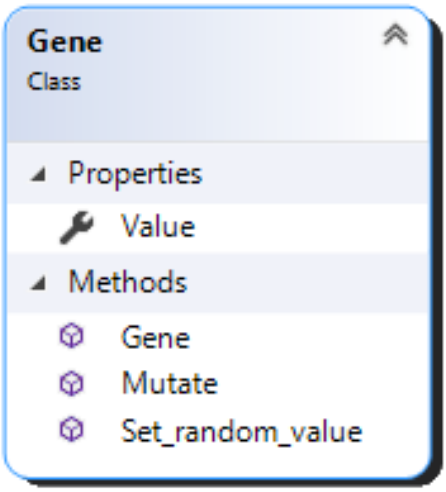


Рисунок 4.19 – Диаграмма класса Gene, показывающая реализацию генов

Выводы по главе 4

В данной главе представлено разработанное программное приложение, описана его структура и основные модули. Подробно рассмотрен базовый интерфейс приложения, его входные и выходные данные.

Разработанное приложение основывается на методах генерации тестовых данных, предложенных во второй и третьей главах диссертации. Оно позволяет генерировать наборы тестовых данных с помощью двух предложенных в диссертации методов. Первый метод обеспечивается многократным запуском генерации данных для одного пути, второй позволяет генерировать множество тестовых наборов одним запуском. Оба метода направлены на достижение максимального покрытия тестируемого кода необходимым для этого количеством тестовых наборов.

Приложение прошло государственную регистрацию в качестве программы для ЭВМ, о чём федеральной службой по интеллектуальной собственности, патентам и товарным знакам выданы свидетельства № 2020663453 от 28 октября 2020 г. и № 2020666236 от 7 декабря 2020 г (Приложение В).

ЗАКЛЮЧЕНИЕ

Основными результатами диссертационной работы являются:

1. Сформулирована общая постановка задачи генерации тестовых данных для ее решения с помощью генетического алгоритма, включая математический вид функции приспособленности, определяющей качество тестового набора. Для вычисления весов функции приспособленности предложены 4 метрики оценки сложности кода, 2 из которых выбраны для дальнейших исследований.

2. Предложены два подхода к генерации множества тестовых наборов, обеспечивающих максимальное покрытие кода, на основе модификации функции приспособленности ГА. Первая модификация заключается во введении дополнительной аддитивной компоненты, отвечающей за разнообразие популяции. Во второй модификации функции приспособленности разнообразие достигается за счет динамического изменения весов операторов в зависимости от степени их покрытия тестовыми наборами, полученными на предыдущей итерации.

3. Разработано семейство алгоритмов генерации тестовых данных на основе предложенных формальных эволюционных постановок задач и различных вариантов функции приспособленности ГА, обеспечивающих максимально возможное покрытие исследуемых программ.

4. Разработано программное приложение, реализующее предложенные алгоритмы генерации тестовых данных. На вход приложения подается текст анализируемой программы, на выходе получается множество сгенерированных тестовых данных с дополнительной информацией о покрытии операторов, значениях функции приспособленности ГА, позволяющей провести анализ качества тестируемого ПО.

5. С использованием разработанного приложения проведен анализ эффективности построенных алгоритмов, найдены значения параметров ГА для повышения скорости сходимости, предложены эвристические решения, повышающие качество тестовых наборов за счет увеличения степени покрытия ими тестируемого кода.

СПИСОК ЛИТЕРАТУРЫ

1. Сердюков, К.Е. Исследование методов определения сложности кода при формировании наборов входных тестовых данных / К. Е. Сердюков, Т. В. Авдеенко // Южно-Сибирский научный вестник. – 2019. – № 4-2 (28). – С. 65–69.
2. Сердюков, К.Е. Исследование метрик оценки кода при формировании наборов данных с использованием генетического алгоритма / К. Е. Сердюков, Т. В. Авдеенко // Известия Тульского государственного университета. Технические науки. – 2019 – Вып. 10 – С. 430–442.
3. Avdeenko, T., Serdyukov, K. Automated Test Data Generation Based on a Genetic Algorithm with Maximum Code Coverage and Population Diversity [Electronic resource] // Applied Sciences. - 2021, 11, 4673. https://doi.org/10.3390/app11104673.
4. Avdeenko, T. V. Genetic algorithm fitness function formulation for test data generation with maximum statement coverage / T. V. Avdeenko, K. E. Serdyukov. - DOI 10.1007/978-3-030-78743-1_34. - Text : direct // Lecture Notes in Computer Science. - 2021. - Vol. 12689 : 12 International Conference on Advances in Swarm Intelligence (ICSI 2021), Qingdao, China, 17 July - 21 July 2021. - P. 379-389.
5. Avdeenko, T.V. Formulation and research of new fitness function in the genetic algorithm for maximum code coverage [Electronic resource] / T. V. Avdeenko, K. E. Serdyukov, Z. B. Tsydenov // Procedia Computer Science. - 2021. - Vol. 186. - P. 713-720. - DOI: 10.1016/j.procs.2021.04.194
6. Serdyukov K.E. Development and Research of the Test Data Generation Approach Modifications / K.E. Serdyukov, T.V. Avdeenko. - DOI 10.1109/ITNT52450.2021.9649110. - Text: direct // The 7 international conference on information technology and nanotechnology (ITNT-2021) : proc., Samara, 20–24 sept. 2021. – Samara : IEEE, 2021. - 6 p.

7. Serdyukov, K. E. Researching of methods for assessing the complexity of program code when generating input test data [Electronic resource] / K. E. Serdyukov, T. V. Avdeenko // CEUR Workshop Proceedings. - 2020. - Vol. 2667: Information Technology and Nanotechnology, ITNT-DS 2020, Samara. - P. 299-304.
8. Serdyukov, K. The Study of the Sequential Inclusion of Paths in the Analysis of Program Code f or the Task of Selecting Input Test Data [Electronic resource] / K. Serdyukov, T. Avdeenko // CEUR Workshop Proceedings. - ata Analytics CEUR Workshop Proceedings. - 2020. - Vol. - 2790: Management in Data Intensive Domains (DAMDID/RCDL 2020), Voronezh, 13-16 Oct. 2020. - P. 79-88.
9. Serdyukov, K. E. Using genetic algorithm for generating optimal data sets to automatic testing the program code / K. E. Serdyukov, T. V. Avdeenko // CEUR Workshop Proceedings. - 2019. – Vol. 2416: Information Technology and Nanotechnology: Data Science, 2019. – P. 173–182.
10. Serdyukov, K. E. Automatic data generation for software testing based on the genetic algorithm / K. E. Serdyukov, T. V. Avdeenko // Actual problems of electronic instrument engineering (APEIE–2018): тр. 14 междунар. науч.-техн. конф., Новосибирск, 2–6 окт. 2018 г.:в 8 т. – Новосибирск : Изд-во НГТУ, 2018. – Т. 1, ч. 4. – С. 535-540.
11. Serdyukov, K. E. Method of application of the genetic algorithm for automatic generation of test data / K. E. Serdyukov, T. V. Avdeenko // CEUR Workshop Proceedings. - 2018. - Vol. 2212: Data Science. Information Technology and Nanotechnology, Samara, 2018. - P. 424-430.
12. Serdyukov, K. E. Investigation of the genetic algorithm possibilities for retrieving relevant cases from big data in the decision support systems / K. E. Serdyukov, T. V. Avdeenko // CEUR Workshop Proceedings. - 2017. - Vol.1903: Data Science. Information Technology and Nanotechnology, DS-ITNT 2017. - P. 36-41.
13. Свидетельство о государственной регистрации программы для ЭВМ №2020663453 Программа генерации тестовых данных на основе

модификации генетического алгоритма с использованием методов оценки сложности кода / К.Е. Сердюков, Т.В. Авдеенко; заяв. и патентообладатель ФГБОУ ВО «Новосибирский государственный технический университет». – №2020662849; заявл. 28.10.2020; зарег. в реестре программ для ЭВМ 28.10.2020 г.

14. Свидетельство о государственной регистрации программы для ЭВМ №2020666236 Программа генерации тестовых данных на основе расширенной функции приспособленности, учитывающей меру сложности кода и разнообразие особей в популяции / К.Е. Сердюков, Т.В. Авдеенко; заяв. и патентообладатель ФГБОУ ВО «Новосибирский государственный технический университет». – №2020665707; заявл. 07.12.2020; зарег. в реестре программ для ЭВМ 07.12.2020 г.
15. Сердюков, К. Е. Разработка и исследование модификаций подхода генерации тестовых данных / К. Е. Сердюков, Т. В. Авдеенко. - Текст : электронный // Информационные технологии и нанотехнологии (ИТНТ-2021), Самара, 20–24 сентября 2021 года - Самара, 2021 – № 33343 – 3 р.
16. Сердюков, К. Е. Исследование способов оценки сложности программного кода при генерации входных тестовых данных = Researching of methods for assessing the complexity of program code when generating input test data / К. Е. Сердюков, Т. В. Авдеенко // Информационные технологии и нанотехнологии (ИТНТ-2020). В 4 т. Т. 4: Науки о данных: сб. тр. 6 междунар. конф. и молодеж. шк., Самара, 26–29 мая 2020 г. – Самара: Изд-во Самар. нац. исслед. ун-та, 2020. – С. 662–671.
17. Сердюков, К. Е. Исследование возможностей применения генетического алгоритма для формирования наборов данных и первичной отладки программного кода / К. Е. Сердюков, Т. В. Авдеенко // Информационные технологии и нанотехнологии (ИТНТ-2019) – Самара : Новая техника, 2019 – Т. 4 – С. 685–694.

18. Сердюков, К. Е. Исследование методов определения сложности кода при формировании наборов входных тестовых данных / К. Е. Сердюков, Т. В. Авдеенко // Измерения, автоматизация и моделирование в промышленности и научных исследованиях (ИАМП–2019) – Бийск: Изд-во АлтГТУ, 2019. – С. 352–355.
19. Сердюков, К. Е. Применение генетического алгоритма для генерации входных данных при тестировании программного кода / К. Е. Сердюков, Т. В. Авдеенко // Наукоемкое программное обеспечение – Новосибирск: Изд-во НГУ, 2019 – С. 130–137.
20. Сердюков, К. Е. Исследование метода применения генетического алгоритма для формирования набора данных при тестировании программного обеспечения / К. Е. Сердюков, Т. В. Авдеенко // Измерения, автоматизация и моделирование в промышленности и научных исследованиях (ИАМП-2018) – Барнаул: Изд-во АлтГТУ, 2018 – С. 528–531.
21. Сердюков, К. Е. Исследование возможностей генетического алгоритма для извлечения релевантных прецедентов в системах поддержки принятия решений / К. Е. Сердюков, Т. В. Авдеенко, Е. С. Макарова // Информационные технологии и нанотехнологии (ИТНТ-2017) – Самара: Изд-во Новая техника, 2017 – С. 1872–1878.
22. Сердюков, К. Е. О возможностях генетического алгоритма для разработки вариантов тестов программного обеспечения / К. Е. Сердюков; науч. рук. Т. В. Авдеенко // Наука. Технологии. Инновации, 2017 г. – Новосибирск: Изд-во НГТУ, 2017 – № 2 – С. 186–190.
23. Сердюков, К. Е. Гибридизация прецедентного подхода к представлению знаний и генетических алгоритмов в экономике / К. Е. Сердюков; науч. рук. Т. В. Авдеенко // Наука. Технологии. Инновации, 2016 г. – Новосибирск: Изд-во НГТУ, 2016 – № 7 – С. 61–63.

24. ISO/IEC/ IEEE 24765: 2010 Systems and Software Engineering – Vocabulary – ISO/ IEC/IEEE, 2010 – 418 p.
25. Липаев, В. Программная инженерия: методологические основы – Direct media, 2015 – 608 с.
26. Антамошкин, О. Программная инженерия. Теория и практика – Litres, 2019 – 320 с.
27. Sommerville, I. Software engineering // Pearson, 9-th edition – 2011 – 790 p.
28. Орлов, С. Технология разработки программного обеспечения – Питер, 2002 – 322 с.
29. ISO/IEC 2382:2015, Information technology — Vocabulary – ISO/IEC, 2015 – Access mode: https://www.iso.org/obp/ui/#iso:std:iso-iec:2382:ed-1:v1:en (access date: 05.02.2021).
30. Lehman, M. On understanding laws, evolution, and conservation in the large-program life cycle – Journal of Systems and Software, 1980 – N 1 – P. 213-221.
31. Голосовский, М. Информационно-логическая модель процесса разработки программного обеспечения – Программные системы и вычислительные методы, 2015 – С. 59-68
32. Budgen, D. Software design // Pearson Education Limited, 2-nd edition – 2003 – 489 p.
33. Рыбалко, М., Иванова Е. Тестирование программного обеспечения, методы тестирования – Кубанский государственный аграрный университет имени И.Т. Трубилина, 2016 – С. 320-322.
34. Lu,, S., Park, S., Seo, E. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics // ASPLOS XIII: Proceedings of the 13th international conference on Architectural support for programming languages and operating systems – 2008 – P. 329–339.

35. Sidorova, T. Static Testing And Dynamic Testing – Understand the Difference – Access mode: https://www.scnsoft.com/software-testing/types-of-bugs (access date: 01.02.2021).
36. Adragna, P. Software debugging techniques // Inverted CERN School of Computing – Computer Science, 2008 – P. 71-86.
37. Grubb, P., Takang, A. Software Maintenance: Concepts and Practice – World Scientific Publishing, 2003 – 350 p.
38. Кулямин, В.В. Методы верификации программного обеспечения – М.: Институт Системного Программирования РАН, 2008 – 111 с.
39. Непомнящий, В. А., Рякин, О. М. Прикладные методы верификации программ. − М.: Радио и связь, 1988. − 255 с.
40. Garanina, N., Anureev, I., Sidorova, E., Koznov, D., Zyubin, V., Gorlatch, S. An Ontology-Based Approach to Support Formal Verification of Concurrent Systems // Lecture Notes in Computer Science, 2020 – N. 12232 LNCS – P. 114-130.
41. Garanina, N., Anureev, I., Borovikova, O., Zyubin, V. Methods for Domain Specialization of Verification-Oriented Process Ontologies // Automatic Control and Computer Sciences, 2020 – N. 54 – P. 740–751.
42. Бурякова, Н.А., Чернов А.В. Классификация частично формализованных и формальных моделей и методов верификации программного обеспечения // Инженерный Вестник Дона – 2010 – № 4 – С. 129-134.
43. Вельдер, С.Э., Шалыто, А.А. О верификации простых автоматных программ на основе метода model checking – Информационно-управляющие системы – № 3 – С. 27-38.
44. Мандрыкин, М.У., Мутилин, В.С., Новиков, Е.М., Хорошилов, А.В. Обзор инструментов статической верификации Си программ в применении к драйверам устройств операционной системы Linux // Сборник трудов Института системного программирования РАН – М.: ИСП РАН, 2012 – Т. 22. – С. 293-294.

45. Шелехов В.И. Дедуктивная верификация и оптимизация предикатной программы конкатенации строк // Системная Информатика – Новосибирск: Институт систем информатики им. А.П. Ершова СО РАН, 2018 – № 12 – С. 61–84
46. Anureev I., Garanina N.O., Liakh T.V., Rozov A.S., Zyubin V.E., Gorlatch S. Two Step Deductive Verification of Control Software Using Reflex // Programming and Computer Software, 2020 – N. 46 (4) – P. 261-272
47. Шелехов В.И. Дедуктивная верификация и реализация предикатной программы инвертирования списков // Информационные и математические технологии в науке и управлении – Иркутск: Институт систем энергетики им. Л.А. Мелентьева СО РАН, 2018 – № 3(11) – С. 136–146
48. Гурин Р. Е., Рудаков И. В., Ребриков А. В. Методы верификации программного – Наука и Образование. М.: МГТУ им. Н.Э. Баумана, 2015 – № 10 – С. 235–251.
49. Graham D., Van Veenendaal E., Evans, I. Foundations of Software Testing – Course Technology Cengage Learning, 2008 – 258 p.
50. Agarwal M. Software Testing Basics: Types of Bugs and Why They Matter – Access mode: https://www.techbeamers.com/static-testing-vs-dynamic-testing/ (access date: 16.03.2021).
51. Static Testing vs Dynamic Testing: What's the Difference? – Access mode: https://www.guru99.com/static-dynamic-testing.html (access date: 16.03.2021).
52. Spillner A., Linz T., Schaefer H. Software Testing Foundations. A Study Guide for the Certified Tester Exam // Rocky Nook – 2014 – 305 p.
53. Korel B. Automated software test data generation // IEEE Transactions on Software Engineering – 1990 – N16 – P. 870–879.
54. Bird D., Munoz C.C. Automatic generation of random self-checking test cases // IBM Systems Journal –1983 – N 22 – P.229–245.

55. Xuan, J., Jiang, H., Ren, Z., Hu, Y., Luo, Z. A random walk based algorithm for structural test case generation // In Proceedings of the 2nd International Conference on Software Engineering and Data Mining – Chengdu, China, 2010 – P. 583-588.
56. Sui, J., Gong, Y., Jin, D., Wang, Y. Statistical testing data generation for UAS // IOP Conference Series: Materials Science and Engineering – 2020 – N 715 – 7 p.
57. Bauer, J., Finger, A. Test plan generation using formal grammars // In Proceedings of the 4th International Conference on Software Engineering – Munich, Germany, 1979 – P. 425–432.
58. Cunning, S.J., Rozenblit, J. Test scenario generation from a structured requirements specification // In Proceedings of the ECBS'99, IEEE Conference and Workshop on Engineering of Computer-Based Systems – Nashville, TN, USA, 1999 – P. 166–172.
59. Doungsa-ard, C., Dahal, K., Hossain A.G., Suwannasart T. An Automatic Test Data Generation from UML State Diagram Using Genetic Algorithm // IEEE Computer Society Press – Washington, DC, USA, 2007 – P. 47–52.
60. Sabharwal, S., Sibal, R., Sharma C. Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams // IJCSI International Journal of Computer Science – 2011 – N 8 – P. 433–444.
61. Doungsa-ard, C., Dahal, K., Hossain, A., Suwannasart, T. GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths. In Advanced Design and Manufacture to Gain a Competitive Edge: New Manufacturing Techniques and Their Role in Improving Enterprise Performance – Springer: London, UK, 2008 – P. 147–156.
62. Grochtmann, M., Grimm, K. Classification trees for partition testing // Software Testing, Verification and Reliability – 1993 – N 3 – P. 63–82.
63. Chen, T.Y., Poon, P.L., Tse, T.H. An integrated classification-tree methodology for test case generation // International Journal of Software Engineering and Knowledge Engineering – 2000 – N 10 – P. 647–679.

64. Cain A., Chen T.Y., Grant D., Poon P.L., Tang S.F., Tse T.H. An Automatic Test Data Generation System Based on the Integrated Classification-Tree Methodology // Software Engineering Research and Applications, Lecture Notes in Computer Science – Springer: Berlin/Heidelberg, Germany, 2004 – N 3026.
65. Bicevskis J., Borzovs J., Straujums U., Zarins A., Miller E. SMOTL-A system to construct samples for data processing program debugging // IEEE Transactions on Software Engineering – 1979 – N SE-5 – P. 60–66.
66. Boyer R., Elspas B., Levitt K. SELECT-A formal system for testing and debugging programs by symbolic execution // ACM SIGPLAN Notices – 1975 – N 10 – P. 234–245.
67. Clarke L. A system to generate test data and symbolically execute programs // IEEE Transactions on Software Engineering – 1976 – N SE-2 – P. 215–222.
68. Howden W. Symbolic testing and the DISSECT symbolic evaluation system // IEEE Transactions on Software Engineering – 1977 – N SE-4 – P. 266–278.
69. Ramamoorthy C.V., Ho S.F., Chen W.T. On the automated generation of program test data // IEEE Transactions on Software Engineering – 1976 – N SE-2 – P. 293–300.
70. Richard A.D., Jefferson A.O. Constraint-Based Automatic Test Data Generation // IEEE Transactions on Software Engineering – 1991 – N 17 – P. 900–910.
71. Meudec C. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution // Software Testing, Verification and Reliability – 2001 – N 11 – P. 81–96.
72. Gerlich R. Automatic Test Data Generation and Model Checking with CHR – arXiv, 2014 – N arXiv:1406.2122 – 8 p.
73. Ferguson R., Korel B. The Chaining Approach for Software Test Data Generation // ACM Transactions on Software Engineering and Methodology – 1996 – N 5 – P. 63–86.

74. Girgis, M.R. Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm // Journal of Universal Computer Science – 2005 – N 11 – P. 898–915.
75. Khamis, A., Bahgat, R., Abdelaziz R. Automatic test data generation using data flow information // Dogus University Journal – 2011 – N 2 – P. 140–153.
76. Liu, Z., Chen, Z., Fang, C., Shi, Q. Hybrid Test Data Generation. State Key Laboratory for Novel Software Technology // In Proceedings of the ICSE Companion 2014 Companion Proceedings of the 36th International Conference on Software Engineering – Hyderabad, India, 2014 – P. 630–631.
77. Harman, M., McMinn, P. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search // IEEE Transactions on Software Engineering – 2010 – N 36 – P. 226–247.
78. Maragathavalli, P., Anusha, M., Geethamalini, P., Priyadharsini, S. Automatic Test-Data Generation for Modified Condition. Decision Coverage Using Genetic Algorithm // International Journal of Engineering Science and Technology – 2011 – N 3(2) – P. 1311–1318.
79. Sharma, A., Patani, R., Aggarwal, A. Software Testing Using Genetic Algorithms // International Journal of Computer Science & Engineering Survey – 2016 – N 7(2) – P. 21–33.
80. Praveen, R.S., Tai-hoon, K. Application of Genetic Algorithm in Software Testing // International Journal of Software Engineering and its Applications – 2009 – N 3(4) – P. 87–96.
81. Berndt, D.J., Watkins, A. Investigating the Performance of Genetic Algorithm-Based Software Test Case Generation // In Proceedings of the Eighth IEEE International Symposium on High Assurance Systems Engineering (HASE'04) – Tampa, FL, USA, 2004 – P. 261–262.

82. Kumar, M., Chaudhary, J. Reviewing Automatic Test Data Generation // International Journal of Engineering Science and Computing – 2017 – N 7 – P. 11432–11435.
83. Bueno, P.M., Wong, W.E., Jino, M. Automatic test data generation using particle systems // In Proceedings of the SAC ‘08: Proceedings of the 2008 ACM symposium on Applied computing – Fortaleza, Brazil, 2008 – P. 809–814.
84. Dixit, S., Tomar, P. Applying Computational Intelligence in Software Testing // Journal of Artificial Intelligence Research & Advances –2015 – N 2(2) – P. 7–11.
85. Li, K., Zhang, Z., Kou, J. Breeding Software Test Data with Genetic-Particle Swarm Mixed Algorithm // Journal of Computers – 2010 – N 5(2) – P. 258–265.
86. Ding, R., Feng, X., Li, S., Dong H. Automatic generation of software test data based on hybrid particle swarm genetic algorithm // In Proceedings of the 2012 IEEE Symposium on Electrical & Electronics Engineering (EEESYM) – Kuala Lumpur, Malaysia, 2012 – P. 670–673.
87. Khan, S.A., Nadeem, A. Automated Test Data Generation for Coupling Based Integration Testing of Object Oriented Programs Using Particle Swarm Optimization (PSO) // In Genetic and Evolutionary Computing. Advances in Intelligent Systems and Computing – Springer: Cham, Switzerland, 2014 – N 238 – P. 115-124.
88. Zhu, E., Yao, C., Ma, Z., Liu, F. Study of an Improved Genetic Algorithm for Multiple Paths Automatic Software Test Case Generation // In Advances in Swarm Intelligence – Springer: Cham, Switzerland, 2017 – P. 402–408.
89. Mirhosseini, S.M., Haghighi, H. A Search-Based Test Data Generation Method for Concurrent Programs // International Journal of Computational Intelligence Systems – 2020 – N 13(1) – P. 1161–1175.
90. Rechenberg, I. Cybernetic Solution Path of an Experimental Problem – Royal Aircraft Establishment Library Translation, 1965 – 784 p.

91. Schwefel, H. P. Understanding evolution as a collective strategy for groping in the dark // Parallelism, Learning Workshop on Evolution and Evolution Models and Strategies – Germany, 1989 – P. 338-398.
92. Simon, D. Evolutionary Optimization Algorithms: Biologically-Inspired and Population-Based Approaches to Computer Intelligence – A John Wiley & Sons Publication, 2013 – 784 p.
93. Holland, J. H. Adaptation in Natural and Artificial Systems – MIT Press Cambridge, 1975 – 236 p.
94. Mitchel, M. An introduction to genetic algorithms – London: A Brad-ford Book The MIT Press, 1999 – 162 p.
95. Koza, J. R. Genetic Programming: On the Programming of Computers by Means of Natural Selection // Statistics and Computing – MIT Press, 1992–N 4 – P. 87-112.
96. De Jong, K. A. Analysis of the behavior of a class of genetic adaptive systems – MIT Press Cambridge, MA, USA – 1992 – 183 p.
97. Davis, L.D. Handbook of Genetic Algorithms–Van Nostrand Reinhold, 1991– 385 p.
98. Haupt, R.L, Haupt, S.E. Practical Genetic Algorithms – 2nd ed. – A John Wiley & Sons Publication, 2004 – 272 p.
99. Holland, J. H. Adaptation in natural and artificial systems: An introductory analysis with applications to biology, control, and artificial intelligence – U Michigan Press, 1975 – 236 p.
100. Barr, K.G. A decimal Gray code – Easily converted for shaft position coding // Wireless World – Faculty of Natural Sciences, University of the West Indies, 1981 – N 87(1542) – P. 86–87.
101. Yang, H., Wang C. Two stages of case-based reasoning // Integrating genetic algorithm with data mining mechanism – Expert Systems with Applications, 2008– N 35 – P. 262–272.

102. Heitkotter, J., Beasly, D. The Hitch-Hiker's Guide to Evolutionary Computation: A List of Frequently Asked Questions (FAQ) – Access mode: ftp://rtfm.mit.edu/pub/usenet/news.answers/ai-faq/genetic/ (access date: 16.06.2021).
103. DeJong, K.A. Learning with Genetic Algorithms: An overview // Machine Learning 3, 1988 – p. 121-138.
104. Graph Optimization via Genetic Algorithm – Access mode: https://www.omgwiki.org/hpec/files/hpec-challenge/ga.html (access date: 12.01.2021).
105. Rajakumar, B.R., George, A. APOGA: An Adaptive Population Pool Size based Genetic Algorithm // AASRI Procedia – Elsevier, 2013 – N 4 – P. 288-296.
106. Riazi, A. Genetic algorithm and a double-chromosome implementation to the traveling salesman problem – SN Applied Sciences 1, 2019 – N 1397.
107. Sivanandam, S.N., Deepa, S. N. Introduction to Genetic Algorithms – Springer, 2008 – 442 p.
108. Carr, J. An Introduction to Genetic Algorithms – Computer Science, 2014.
109. Jiang, C., Serrao P., Liu M., Cho C. An Enhanced Genetic Algorithm for Parameter Estimation of Sinusoidal Signals – Applied Science, 2020 – N 10(15) – 16 p.
110. Dorigo, M., Birattari, M., Stutzle, T. Ant colony optimization – IEEE Computational Intelligence Magazine, 2006 – N 1(4) – P. 28–39.

ПРИЛОЖЕНИЕ А ЛИСТИНГИ ТЕСТИРУЕМЫХ ПРОГРАММ

Листинг тестируемой программы первой вариации SUT1

```
using System;
using System.Collections.Generic;
public class TestClass
{
    int result_value = 0;

    public TestClass() { }

    public void Main(int val1, int val2, int val3)
    {
        int current_weight_height = 100;
        int current_weight_low = 0;
        List<int> weight_height = new List<int>();
        List<double> weight_low = new List<double>();
        List<double> experement = new List<double>();
        double var1 = 0;
        double var2 = 0;
        double var3 = 0;
        int weight_count = 0;
        weight_height.Add(100);
        weight_low.Add(0);
        for (int i = 0; i < 100; i++)
        {
            if ((val1 > 5 && val1 < 60) || (val2 > 90 || val2 == 10))
            {
                var1 = Math.Abs(val1);
                var2 = Math.Log(val2);
                experement.Add(var1 + var2);
            }
            else if (val1 == 60)
            {
                current_weight_height = Convert.ToInt32(weight_height[weight_count] * 0.8);
                current_weight_low = weight_height[weight_count] - current_weight_height;
                weight_height.Add(current_weight_height);
                weight_low.Add(current_weight_low);
                weight_count += 1;
                var1 = current_weight_height;
                var2 = current_weight_low;
                experement.Add(var1 + var2);
            }
            else if (val3 > 5 && val1 < 30 && weight_count > 1)
            {
                var1 = weight_height[weight_count];
                weight_height.RemoveAt(weight_count);
                var2 = weight_low[weight_count];
                weight_low.RemoveAt(weight_count);
                weight_count -= 1;
                experement.Add(var1 + var2);
            }
            else if (val3 > 50)
            {
                if (val1 < val3)
                {
                    int count_condition_blocks = 1;
                    int divider = 1;
```

```
                    int n = 0;
                    while (n < 10)
                    {
                        var1 = val1;
                        var2 = Math.Exp(val2);
                        experement.Add(var1 * var2);
                        n += 1;
                    }
                    var3 = 50 - var2;
                    if (val1 > 50)
                    {
                        divider = count_condition_blocks;
                        current_weight_height = Convert.ToInt32(weight_height[weight_count]
/ divider);
                        current_weight_low = current_weight_height;
                        weight_height.Add(current_weight_height);
                        weight_low.Add(current_weight_low);
                        weight_count += 1;
                        var1 = Math.Abs(current_weight_height);
                        var2 = Math.Abs(current_weight_low);
                        experement.Add(var1 + var2);
                    }
                    else
                    {
                        divider = Convert.ToInt32(count_condition_blocks + 0.25);
                        current_weight_height = Convert.ToInt32(weight_height[weight_count]
/ divider);
                        current_weight_low = weight_height[weight_count] -
current_weight_height * count_condition_blocks;
                        weight_height.Add(current_weight_height);
                        weight_low.Add(current_weight_low);
                        weight_count += 1;
                        var1 = Math.Abs(current_weight_height);
                        var2 = Math.Abs(current_weight_low);
                        experement.Add(var1 + var2);
                    }
                }
            }

        }
        for (int i = 0; i < experement.Count; i++)
        {
            Console.WriteLine(experement[i]);
        }
    }
}
```

Листинг тестируемой программы второй вариации SUT1

```
using System;
using System.Collections.Generic;
public class TestClass
{
    int result_value = 0;

    public TestClass() { }

    public void Main(int val1, int val2, int val3)
```

```
    {
        int current_weight_height = 100;
        int current_weight_low = 0;
        List<int> weight_height = new List<int>();
        List<double> weight_low = new List<double>();
        List<double> experement = new List<double>();
        double var1 = 0;
        double var2 = 0;
        double var3 = 0;
        int weight_count = 0;
        weight_height.Add(100);
        weight_low.Add(0);
        for (int i = 0; i < 100; i++)
        {
            if ((val1 > 5 && val1 < 60) || (val2 > 90 || val2 == 10))
            {
                var1 = Math.Abs(val1);
                var2 = Math.Log(val2);
                experement.Add(var1 + var2);
            }
            else if (val1 == 60)
            {
                current_weight_height = Convert.ToInt32(weight_height[weight_count] * 0.8);
                current_weight_low = weight_height[weight_count] - current_weight_height;
                weight_height.Add(current_weight_height);
                weight_low.Add(current_weight_low);
                weight_count += 1;
                var1 = current_weight_height;
                var2 = current_weight_low;
                experement.Add(var1 + var2);
            }
            else if (val3 > 5 && val1 < 30 && weight_count > 1)
            {
                var1 = weight_height[weight_count];
                weight_height.RemoveAt(weight_count);
                var2 = weight_low[weight_count];
                weight_low.RemoveAt(weight_count);
                weight_count -= 1;
                experement.Add(var1 + var2);
            }
            else if (val3 > 50)
            {
                if (val1 < val3)
                {
                    int count_condition_blocks = 1;
                    int divider = 1;
                    int n = 0;
                    while (n < 10)
                    {
                        var1 = val1;
                        var2 = Math.Exp(val2);
                        experement.Add(var1 * var2);
                        n += 1;
                    }
                    var3 = 50 - var2;
                    if (val1 > 50)
                    {
                        divider = count_condition_blocks;
                        current_weight_height = Convert.ToInt32(weight_height[weight_count]
/ divider);
```

```
                        current_weight_low = current_weight_height;
                        weight_height.Add(current_weight_height);
                        weight_low.Add(current_weight_low);
                        weight_count += 1;
                        var1 = Math.Abs(current_weight_height);
                        var2 = Math.Abs(current_weight_low);
                        experement.Add(var1 + var2);
                    }
                    else
                    {
                        divider = Convert.ToInt32(count_condition_blocks + 0.25);
                        current_weight_height = Convert.ToInt32(weight_height[weight_count]
/ divider);
                        current_weight_low = weight_height[weight_count] -
current_weight_height * count_condition_blocks;
                        weight_height.Add(current_weight_height);
                        weight_low.Add(current_weight_low);
                        weight_count += 1;
                        var1 = Math.Abs(current_weight_height);
                        var2 = Math.Abs(current_weight_low);
                        experement.Add(var1 + var2);
                    }
                }
                else if (val2 == val3)
                {
                    current_weight_height = weight_height[weight_count];
                    if (val2 > 50)
                    {
                        current_weight_low = weight_height[weight_count];
                    }
                    else
                    {
                        current_weight_low = Convert.ToInt32(weight_low[weight_count]);
                    }
                    var1 = Math.Abs(current_weight_height);
                    var2 = Math.Abs(current_weight_low);
                    experement.Add(var1 + var2);
                }
                else
                {
                    current_weight_height = weight_height[weight_count];
                    var3 = current_weight_height;
                }
            }
            else if (val1 > 60 && val2 > 55 && weight_count > 1)
            {
                weight_height.RemoveAt(weight_count);
                weight_low.RemoveAt(weight_count);
                weight_count -= 1;
                var1 = weight_height[weight_count];
                if (val2 > 90)
                {
                    var1 = Math.Sin(var3);
                    var1 = Math.Cos(var3);
                    experement.Add(var1 * var2);
                }
            }
        }
        for (int i = 0; i < experement.Count; i++)
        {
```

```
            Console.WriteLine(experement[i]);
        }
    }
}
```

Листинг тестирумой программы SUT2

```
using System;
using System.Collections.Generic;
using System.Text;
public class TestClass
{
    int result_value = 0;
    public TestClass() { }
    public void Main(int executionNumber, int val1, int val2, int val3, int val4, int val5)
    {
        Console.WriteLine("Initial Values " + val1 + "|" + val2 + "|" + val3 + "|" + val4 +
"|" + val5 + ".");
        for (int i = 0; i < executionNumber; i++)
        {
            if (IsPalindromeNumber(val1) || IsPalindromeNumber(val2) ||
IsPalindromeNumber(val3) || IsPalindromeNumber(val4) || IsPalindromeNumber(val5))
            {

                if (val1 == Largest4Number(val2, val3, val4, val5))
                {
                    string transformNumber1 = RomanNumberTranslate(val1);
                    string transformNumber2 = RomanNumberTranslate(val2);
                    string transformNumber3 = RomanNumberTranslate(val3);
                    string transformNumber4 = RomanNumberTranslate(val4);
                    string transformNumber5 = RomanNumberTranslate(val5);
                    Console.WriteLine("Transformed to Roman: " + transformNumber1 + "|" +
transformNumber2 +"|"+ transformNumber3 + "|" + transformNumber4 + "|" + transformNumber5 );
                    if (val2 <= val4 / 2)
                    {
                        if (IsOddNumber(val2) && IsOddNumber(val4))
                        {
                            Console.WriteLine("GCD First: " + GetGcd(val2, val4));
                            Console.WriteLine("HCF First: " + GetHcf(val2, val4));
                            Console.WriteLine("GCD Second: " + GetGcd(val4, val2));
                            Console.WriteLine("HCF Second " + GetHcf(val4, val2));
                        }
                    }
                }
                if (IsPrimeNumber(val1) && IsPrimeNumber(val3) && IsPrimeNumber(val5))
                {
                    if (NumberMoreThan(val1, val3) && NumberMoreThan(val1, val5))
                    {
                        Console.WriteLine("From example values there is prime number and " +
val1 + " is maximum.");
                        val1 = SumOfNumbers(val1, val3, val5);
                        Console.WriteLine("New value is " + val1);
                        if (val1 < val2 && val1 < val4)
                        {
                            val1 = SumOfNumbers(val1, val2, val3, val4, val5);
                        }
                    }
                    else if (NumberLessThan(val3, val5))
```

```
                    {
                        Console.WriteLine("Cannot find maximum value. Decrease values.");
                        if (val1 > 0)
                        {
                            val1 = val1 / SumOfNumbers(val1) * 2;
                        }
                        if (val3 > 0)
                        {
                            val3 = val3 / SumOfNumbers(val3) * 2;
                        }
                        if (val5 > 0)
                        {
                            val5 = val5 / SumOfNumbers(val5) * 2;
                        }
                    }
                }
            }
            if (val2 > val3 && val1 < val5)
            {
                int checkDouble = 1;
                int max = Largest4Number(val1, val2, val3, val4, val5);
                while (checkDouble <= max)
                {
                    checkDouble += 1;
                    if (val1 < val3)
                    {
                        if (checkDouble % 2 == 0)
                        {
                            val3 = ReverseNumber(val3);
                        }
                        else
                        {
                            val1 = ReverseNumber(val1);
                        }
                    }
                    else
                    {
                        if (IsAmicableNumbers(val1, val3))
                        {
                            Console.WriteLine("On the " + checkDouble + " iteration first
and third values is amicable");
                            break;
                        }
                        else
                        {
                            Console.WriteLine("On the " + checkDouble + " iteration first
and third values is NOT amicable");
                            break;
                        }
                    }
                    Console.WriteLine("Inner Cycle, value: "+checkDouble);
                }
                if (val3 < val2)
                {
                    Console.WriteLine("Divide third value by 2");
                    val2 = Divide(val2, 2);
                }
                if (val1 < val5)
                {
                    Console.WriteLine("Divide first value by 2");
```

```
                    val5 = Divide(val5, 2);
                }
                if (val3 < val2 && val1 < val5)
                {
                    int lcm1 = LCM(val3, val2);
                    int lcm2 = LCM(val5, val1);
                    Console.WriteLine("LCM1 = " + lcm1 + " | LCM2 = " + lcm2);
                }
            }
            else
            {
                if (Power(val3, 2) < val2)
                {
                    Console.WriteLine("Power third value by 2");
                    val3 = Power(val3, 2);
                }
                else
                {
                    val2 -= SumOfNumbers(val2);
                }
                if (Power(val1, 2) < val5)
                {
                    Console.WriteLine("Power first value by 2");
                    val1 = Power(val1, 2);
                }
                else
                {
                    val5 -= SumOfNumbers(val5);
                }
            }
            Console.WriteLine("Outer Cycle, iteration: " + executionNumber);
        }
        if (executionNumber < 100)
        {
            MessageReturn(executionNumber);
        }
        else
        {
            Console.WriteLine("Too many executions");
        }
    }
    public int SumOfNumbers(int val1, int val2 = 0, int val3 = 0, int val4 = 0, int val5 =
0)
    {
        int tval1 = val1;
        int tval2 = val2;
        int tval3 = val3;
        int tval4 = val4;
        int tval5 = val5;
        int sum = 0;
        int r = 0;
        while (tval1 != 0)
        {
            r = tval1 % 10;
            tval1 = tval1 / 10;
            sum += r;
        }
        while (tval2 != 0)
        {
            r = tval2 % 10;
```

```
            tval2 = tval2 / 10;
            sum += r;
        }
        while (tval3 != 0)
        {
            r = tval3 % 10;
            tval3 = tval3 / 10;
            sum += r;
        }
        while (tval4 != 0)
        {
            r = tval4 % 10;
            tval4 = tval4 / 10;
            sum += r;
        }
        while (tval5 != 0)
        {
            r = tval5 % 10;
            tval5 = tval5 / 10;
            sum += r;
        }
        return sum;
    }
    public bool IsPrimeNumber(int val)
    {
        int m = val / 2;
        bool flag = false;
        bool result = false; ;
        for (int i = 2; i <= m; i++)
        {
            if (val % i == 0)
            {
                result = false;
                flag = true;
                break;
            }
        }
        if (!flag)
        {
            result = true;
        }
        return result;
    }
    public bool IsOddNumber(int val)
    {
        if (val % 2 == 0)
        {
            return false;
        }
        else
        {
            return true;
        }
    }
    public bool IsPalindromeNumber(int val)
    {
        if (val < 10)
        {
            return false;
        }
```

```
        int reverseNumber = 0, temp, rem;
        temp = val;
        while (temp != 0)
        {
            rem = temp % 10;
            temp = temp / 10;
            reverseNumber = reverseNumber * 10 + rem;
        }
        if (val == reverseNumber)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    public bool NumberMoreThan(int val1, int val2)
    {
        if (val1 >= val2)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    public bool NumberLessThan(int val1, int val2)
    {
        if (val1 < val2)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
    public int ReverseNumber(int val1)
    {
        int a = val1, rev = 0, b;
        while (a != 0)
        {
            b = a % 10;
            rev = (rev * 10) + b;
            a = a / 10;
        }
        return rev;
    }
    public string RomanNumberTranslate(int val1)
    {
        string[] roman_symbol = { "MMM", "MM", "M", "CM", "DCCC", "DCC", "DC", "D", "CD",
"CCC", "CC", "C", "XC", "LXXX", "LXX", "LX", "L", "XL", "XXX", "XX", "X", "IX", "VIII",
"VII", "VI", "V", "IV", "III", "II", "I" };
        int[] int_value = { 3000, 2000, 1000, 900, 800, 700, 600, 500, 400, 300, 200, 100,
90, 80, 70, 60, 50, 40, 30, 20, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };
        int test_value = Math.Abs(val1);
        var roman_numerals = new StringBuilder();
        var index_num = 0;
```

```
        while (test_value != 0)
        {
            if (test_value >= int_value[index_num])
            {
                test_value -= int_value[index_num];
                roman_numerals.Append(roman_symbol[index_num]);
            }
            else
            {
                index_num++;
            }
        }
        return roman_numerals.ToString();
    }
    public int Divide(int dividend, int divisor)
    {
        uint x = dividend > 0 ? (uint)dividend : (uint)-dividend;
        uint y = divisor > 0 ? (uint)divisor : (uint)-divisor;
        uint result = 0, z = 0;
        var idx = 0;
        while (x >= y)
        {
            z = y;
            for (idx = 0; x >= z && z != 0; idx++, z *= 2)
            {
                x -= z;
                result += (uint)1 << idx;
            }
        }
        int res = 0;
        if ((dividend ^ divisor) >> 31 == -1)
        {
            res = (int)-result;
        }
        else
        {
            if (result > int.MaxValue)
            {
                res = int.MaxValue;
            }
            else
            {
                res = (int)result;
            }
        }
        return res;
    }
    public int Power(int number, int power)
    {
        if (power == 0)
        {
            return 1;
        }
        else
        {
            return number * Power(number, power - 1);
        }
    }
    public int Largest4Number(int val1, int val2, int val3 = 0, int val4 = 0, int val5 = 0)
    {
```

```
        int large = val1;
        if (val1 > val2 && val1 > val3 && val1 > val4 && val1 > val5)
        {
            large = val1;
        }
        else if (val2 > val1 && val2 > val3 && val2 > val4 && val2 > val5)
        {
            large = val2;
        }
        else if (val3 > val1 && val3 > val2 && val3 > val4 && val3 > val5)
        {
            large = val3;
        }
        else if (val4 > val1 && val4 > val2 && val4 > val3 && val4 > val5)
        {
            large = val4;
        }
        else
        {
            large = val5;
        }
        return large;
    }
    public bool IsAmicableNumbers(int val1, int val2)
    {
        int sum1 = 0;
        int sum2 = 0;
        int X = 0;
        for (X = 1; X < val1; X++)
        {
            if (val1 % X == 0)
            {
                sum1 = sum1 + X;
            }
        }
        for (X = 1; X < val2; X++)
        {
            if (val2 % X == 0)
            {
                sum2 = sum2 + X;
            }
        }
        if (val1 == sum2 && val2 == sum1)
        {
            return true;
        }
        return false;
    }
    public int GetHcf(int val1, int val2)
    {
        int iLoop = 1;
        int hcf = 0;
        while (iLoop <= val1 || iLoop <= val2)
        {
            if (val1 % iLoop == 0 && val2 % iLoop == 0)
            {
                hcf = iLoop;
            }
            iLoop++;
        }
```

```
        return hcf;
    }
    public int GetGcd(int val1, int val2)
    {
        int rem = 0;
        while (val2 > 0)
        {
            rem = val1 % val2;
            val1 = val2;
            val2 = rem;
        }
        return val1;
    }
    public int LCM(int val1, int val2)
    {
        int firstNumber = 0;
        int secondNumber = 0;
        int temp1 = 0;
        int temp2 = 0;
        firstNumber = val1;
        secondNumber = val2;
        if (firstNumber > secondNumber)
        {
            temp1 = firstNumber; temp2 = secondNumber;
        }
        else
        {
            temp1 = secondNumber; temp2 = firstNumber;
        }

        for (int i = 1; i <= temp2; i++)
        {
            if ((temp1 * i) % temp2 == 0)
            {
                return i * temp1;
            }
        }
        return temp2;
    }
    public string MessageReturn(int messageVal)
    {
        if (messageVal < 10)
        {
            return "Less than 10 runs";
        }
        else if (messageVal < 50)
        {
            return "Less than 50 runs";
        }
        else if (messageVal < 90)
        {
            return "Less than 90 runs";
        }
        else
        {
            return "More than 90 runs";
        }
    }
}
```

ПРИЛОЖЕНИЕ Б ФРАГМЕНТЫ ЛИСТИНГА РАЗРАБОТАННОГО ПРИЛОЖЕНИЯ

Main

```
namespace Multi_Test_Data_Generator
{
    /// <summary>
    /// Interaction logic for MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        ...
        private void Data_generation(bool new_run = true, int ga_type = 1, int runs_count =
1)
        {
            ...

            if (compiled)
            {
               //Запуск алгоритма генерации данных в новом потоке
                Thread thread = new Thread(delegate () {
Data_generation_second_step(gen_cnt, pop_cnt, mut_chance, min_val, max_val, ga_type,
runs_count); });
                thread.Start();
            }
        }

        private void Data_generation_second_step(int gen_cnt, int pop_cnt, int mut_chance,
int min_val, int max_val, int ga_type = 1, int runs_count = 1)
        {
            ...
            for (int run = 0; run < runs_count; run++)
            {
                //Установка ГА с переданными параметрами и начало генерации данных
                GA.Set_GA(gen_cnt, pop_cnt, Code_handler.Number_of_params, mut_chance,
min_val, max_val);
                double k_value = Convert.ToDouble(Dispatcher.Invoke(() => { return
k_value_box.Text; }));
                GA.Begin(ga_type, k_value);
                ...
                }

            }
           //Обработка интерфейса для вывода сгенерированных наборов
            ...
        }
        ...
    }
}
```

Code handler

```
namespace Multi_Test_Data_Generator.Code_Handler
{
    public static class Code_handler
```

```
    {

        ...

        /// <summary>
        /// Компиляция кода
        /// </summary>
        /// <param name="code">Исходный код</param>
        /// <param name="metric">NOD, SLOC, ABC, Jilb</param>
        /// <param name="next_run">Истина, если необходимо сохранить наборы между
запусками</param>
        /// <returns></returns>
        public static string Compile(string code, string metric, bool next_run = false)
        {
            //Сохранение наборов между запусками
            if (!next_run || input_text == "" || input_text != code)
            {
                remove_path = new List<int>();
            }
            str_path = new List<string>();
            ...
            //При успешной компиляции кода возвращается сообщение об этом
            return "Code compiled successfully";
        }

        /// <summary>
        /// Обработка весов и выбранных метрий
        /// </summary>
        /// <param name="inputString">Исходный кода для генерации данных</param>
        /// <param name="metric"></param>
        /// <returns></returns>
        private static string Assign_weights(string inputString, string metric, string code)
        {
            ...
            //app_if используется для учёта уровня вложенности операторов
            int app_if = -1;
            List<string> resultStrings = new List<string>(); //Строка с обработанным кодом
            ...
            string[] dividedString = Regex.Split(inputString, @"(\{|\}|\;)");
            foreach (string str in dividedString)
            {
                ...
                //Обработка зависит от выбранной метрики
                if (choosen_metric == 1)
                {
                    //Проверка, что оператор находится внутри функции, что не установлен
флаг на запрет проверки и что до этого не встречалось процедур предотвращения работы
                    if (app_if > 0 && !dont_check && !(past_str.Trim().ToLower() == "break"
|| past_str.Trim().ToLower().StartsWith("return ")))
                    {
                        //Обработка оператора предотвращения работы
                        if (str.Trim().ToLower() == "break" ||
str.Trim().ToLower().StartsWith("return "))
                        {
                            weight_array.Add(oper_number, Convert.ToInt32(weight / app_if));
                        }
                        //Если строка содержит ;, то она считается оператором
                        else if (str.Contains(";"))
                        {
                            weight_array.Add(oper_number, Convert.ToInt32(weight / app_if));
```

```
                        }
                    }

                }
                if (choosen_metric == 2)
                {
                    //Обработка схожа с первой метрикой, но вес будет равен 1
                    ...
                }
                if ((choosen_metric == 3 || choosen_metric == 4) && app_if > 1)
                {

                    ...
                }
            }
            //Количество операторов сохраняется для последующего вычисления покрытия
            all_operation_count = oper_number;
            return string.Join("", resultStrings.ToArray());
        }

        private static Assembly CompileSourceCodeDom(string sourceCode, out string errors)
        {
            //Дополнительная функция для компиляции кода
            ...
        }
        public static int Get_function_value(int[] parameters)
        {
            //Только для успешно скомпилированного кода
            if (!(ga_code_assembly is null))
            {
                ...
                //Определение количества повторений покрытия операторов
                foreach (int operation_number in val_list)
                {
                    //Если оператор есть в массиве, то добавляется 1, иначе назначается 1
                    if (repeat_array.ContainsKey(operation_number))
                    {
                        repeat_array[operation_number] += 1;
                    }
                    else
                    {
                        repeat_array.Add(operation_number, 1);
                    }
                    //Каждый оператор добавляет вес к итоговой сумме
                    if (weight_operations_coef.Count > 0 &&
weight_operations_coef.ContainsKey(operation_number))
                    {
                        //Сумма высчитывается умножением ранее рассчитанного веса и его
коэффициента, определяемый используемой модификацией
                        weight_sum +=
Convert.ToDecimal(weight_array[operation_number])*weight_operations_coef[operation_number];
                    }

                }
                return Convert.ToInt32(weight_sum);
            }
            //Если код не был скомпилирован, то возвращается 0
            return 0;
        }
```

```
        public static void New_generation_update(int crm_count)
        {
            //Функция для перерасчёта коэффициента веса при формировании новой популяции
            ...
        }

        public static void Generations_update(int gen_count, int pop_size = 0)
        {
            //Функция для перерасчёта веса после вычисления функции приспособленности
            ...
        }

        ...
    }
}
```

GA

```
namespace Multi_Test_Data_Generator.Genetic_algorithm
{
    static class GA
    {
        public static List<Population> populations = new List<Population>();

        private static bool GA_set = false; //Флаг установки ГА

        public static int Generation_count { get; private set; }
        public static int Population_size { get; private set; }
        public static int Chromosome_size { get; private set; }
        public static int Mutation_chance { get; private set; }

        public static int Generator_min_value { get; private set; }
        public static int Generator_max_value { get; private set; }

        public static void Set_GA(int gen_number, int pop_number, int chr_number, int
mut_chance,  int min_val, int max_val)
        {
           //Назначение параметров ГА
            ...
        }

        //Основная функция, реализующая цикл генетического алгоритма
        public static void Begin(int ga_type = 1, double k_value = 10)
        {
            ...
            for (int gen_ind = 1; gen_ind <= Generation_count; gen_ind++)
            {
                //Добавление новой пустой популяции
                populations.Add(new Population(Population_size, Chromosome_size));
                //Перенос пула элитных хромосом между популяциями
                if (ga_type == 2)
                {
                    populations[gen_ind].unique_chromosomes = populations[gen_ind -
1].unique_chromosomes.ToList();
                    populations[gen_ind].code_coverage_by_unique = populations[gen_ind -
1].code_coverage_by_unique;
                }
                //20% новой популяции будет получено перенесением из предыдущей
                for (int pop_ind = 0; pop_ind < Population_size/5; pop_ind++)
```

```
                {
                    populations[gen_ind].Replace_chromosomes(pop_ind, populations[gen_ind -
1].chromosomes[pop_ind]);
                }
                for (int pop_ind = 0; pop_ind < Population_size; pop_ind++)
                {
                    //Выбор хромосом для скрещивания и запуск соответствующей функции
                    ...
                }
                //Запуск функции для вычисления функции приспособленности
                populations[gen_ind].Refresh(ga_type, k_value);
                Code_handler.Generations_update(Generation_count, Population_size);
            }
            ...
        }

        //Функция для заполнения популяции случайными хромосомами
        private static void Create_random_population(int ga_type = 1, double k_value = 10)
        {
            ...
        }
    }
}
```

Population

```
namespace Multi_Test_Data_Generator.Genetic_algorithm
{
    class Population
    {
        public List<Chromosome> chromosomes { get; private set; } //Набор хромосом
        public List<Chromosome> unique_chromosomes; //Пул элитных хромосом
        public string code_coverage_by_unique = ""; //Покрытие кода элитными хромосомами
        private readonly int size;
        public double avg_sim_value;
        public double max_sim_value;
        public int code_coverage = 0; //Значение покрытия кода

        //Установка новой пустой популяции
        public Population(int size, int chromosome_size)
        {
            ...

        }

        //Заполнение популяции случайными хромосомами
        public void Fill_random(int min_value, int max_value)
        {
            ...
        }

        //Функция для переноса хромосом без изменений
        public void Replace_chromosomes(int index, Chromosome chr)
        {
            Chromosome tmp_chr = new Chromosome(chr);
            chromosomes[index] = tmp_chr;
```

```
        }

        //Скрещивание хромосом по их индексам
        public Chromosome Crossover_chromosomes(int index_1, int index_2)
        {
            return chromosomes[index_1].Crossover(chromosomes[index_2]);
        }

        //Сортировка популяции по значению функции приспособленности
        public void Sort()
        {
            chromosomes = chromosomes.OrderByDescending(o => o.Fitness_function).ToList();
        }

        //Перерасчёт функции приспособленности для всех хромосом
        public void Refresh(int ga_type = 1, double k_input = 10)
        {
            Code_Handler.Code_handler.New_generation_update(this.chromosomes.Count);
            //Для каждой хромосомы запускается функция вычисления значения приспособленности
            foreach (Chromosome crm in chromosomes)
            {
                crm.Get_fitness_function();
            }
            //Для метода с двумя компонентами дополнительно обрабатывается вторая компонента
            if (ga_type == 2)
            {
                //Рассчитывается среднее значение сходства хромосом в популяции
                Com_avg_sim_value();
                //Определяется словарь для учёта элитных хромосом
                Dictionary<string, int> path_counts = new Dictionary<string, int>();
                foreach (Chromosome crm in chromosomes)
                {
                    double k = k_input;
                    //Если путь ещё не встречался, то добавляем его
                    if (!path_counts.ContainsKey(crm.path_bit_string))
                    {
                        path_counts.Add(crm.path_bit_string, 1);
                    }
                    //Если путь уже покрывался, то увеличиваем значение на 1
                    else
                    {
                        path_counts[crm.path_bit_string] += 1;
                    }
                    //Если пул элитных хромосом пустой, то добавляем самую первую хромосому
                    if (unique_chromosomes.Count == 0)
                    {
                        unique_chromosomes.Add(crm);
                        code_coverage_by_unique = crm.path_bit_string;
                    }
                    crm.first_term = crm.Fitness_function;
                    //Рассчитывается вторая компонента
                    crm.second_term = k * Math.Abs(avg_sim_value - crm.sim_value) /
path_counts[crm.path_bit_string];
                    crm.Fitness_function = crm.first_term + crm.second_term;
                    //Проверка наличия хромосомы в пуле элитных и добавляем её, если она
уникальная
                    if (!Covered_by_others(crm))
                    {
                        unique_chromosomes.Add(crm);
```

```
                        code_coverage_by_unique =
Combine_path_bit_strings(code_coverage_by_unique, crm.path_bit_string);
                    }
                }
            }
            Sort();
            //Происходит вычисление покрытия для двух методов генерации данных
            if (ga_type == 2)
            {
                List<int> path_temp = new List<int>();
                foreach (Chromosome crm in unique_chromosomes)
                {
                    for (int i = 0; i < crm.path.Count; i++)
                    {
                        if (!path_temp.Contains(crm.path[i]))
                        {
                            path_temp.Add(crm.path[i]);
                        }
                    }
                }
                code_coverage = path_temp.Count * 100 /
Code_Handler.Code_handler.all_operation_count;
            }
            else
            {
                List<int> path_temp = Code_Handler.Code_handler.remove_path;
                for (int i = 0; i < chromosomes[0].path.Count; i++)
                {
                    if (!path_temp.Contains(chromosomes[0].path[i]))
                    {
                        path_temp.Add(chromosomes[0].path[i]);
                    }
                }
                code_coverage = path_temp.Count * 100 /
Code_Handler.Code_handler.all_operation_count;
            }

        }

        //Функция для вычисления среднего значения сходства
        public void Com_avg_sim_value()
        {
            ...
        }

        //Функция для объединения строки покрытия операторов
        private string Combine_path_bit_strings(string path1, string path2)
        {
            ...
        }

        /// <summary>
        /// Проверяется неразличимость хромосом
        /// </summary>
        /// <param name="crm"></param>
        /// <returns></returns>
        private bool Covered_by_others(Chromosome crm)
        {
            ...
        }
```

```
    }
}
```

Chromosome

```
namespace Multi_Test_Data_Generator.Genetic_algorithm
{
    class Chromosome
    {
        public List<Gene> genes { get; private set; }
        public readonly int size;

        public bool empty_chromosome; //Флаг пустой хромосомы

        public double Fitness_function { get; set; }
        public double first_term;
        public double second_term;
        public List<int> path = new List<int>();
        /// <summary>
        /// Путь, инициированный хромосомой, в виде битовой строки
        /// </summary>
        public string path_bit_string = "";
        public double sim_value;
        public string add_info = "";

        //Установка пустой хромосомы
        public Chromosome(int size)
        {
            ...
        }

        //Хромосома формируется на основе другой хромосомы
        public Chromosome(Chromosome chr)
        {
            genes = new List<Gene>();
            this.size = chr.size;
            empty_chromosome = false;
            this.genes = chr.genes;
        }

        //Заполнение случайными генами
        public void Fill_random_genes(int min_value, int max_value, Random rnd)
        {
            ...
        }

        //Вычисление функции приспособленности и установка покрываемого пути
        public void Get_fitness_function()
        {
            Fitness_function = Code_handler.Get_function_value(Get_values_array());
            path = Code_handler.val_list;
            path_bit_string = Get_path_bit_string();
        }

        //Перевод пути из массива в битовую строку
        private string Get_path_bit_string()
        {
            ...
```

```
        }

        //Получение значения сходства на основе битовой строки пути
        public int Get_sim_value(string new_bit_string)
        {
            ...
        }

        //Получения значение сходства для двух хромосом
        public int Get_sim_value(Chromosome chr)
        {
            ...
        }

        //Получение значений хромосомы в виде массива
        private int[] Get_values_array()
        {
            ...
        }

        //Обработка эволюционной операции скрещивания
        public Chromosome Crossover(Chromosome crm)
        {
            ...
            Random rnd_cross = new Random(rnd_seed);
            Chromosome new_chromosome = new Chromosome(size);
            //Определение случайного шанса для скрещивания
            int rand_position;
            int rand_position_s = 50;
            //Использование механизма смешивания значений в совокупности со скрещиванием
            for (int b = 0; b < genes.Count; b++)
            {
                //Определяется, какая из хромосом имеет большее значение гена
                if (this.genes[b].Value <= crm.genes[b].Value)
                {
                    if (this.genes[b].Value < 0 && crm.genes[b].Value < 0)
                    {
                        //Смешивание значений, если оба гена отрицательные
                        new_chromosome.genes[b].Value = -
rnd_cross.Next(Math.Abs(crm.genes[b].Value) * 90, Math.Abs(this.genes[b].Value) * 110) /
100;
                    }
                    else
                    {
                        //Смешивание значений, если оба гена положительные
                        new_chromosome.genes[b].Value = rnd_cross.Next(this.genes[b].Value *
90, crm.genes[b].Value * 110) / 100;
                    }
                }
                else
                {
                    //Повторяет реализацию, но в обратную сторону
                    ...
                }

                //Мутация с определённым шансом
                if (rnd_cross.Next(0, 100) <= GA.Mutation_chance)
                {
                   new_chromosome.genes[b].Mutate(rnd_cross);
                }
```

```
            }
            new_chromosome.empty_chromosome = false;
            return new_chromosome;
        }
    }
}
```

Gene

```
namespace Multi_Test_Data_Generator.Genetic_algorithm
{
    class Gene
    {
        public int Value { get; set; } //Значение гена

        public Gene(int value)
        {
            this.Value = value;
        }

        //Мутация, то есть изменение значения гена на другое
        public void Mutate(Random rnd_mut)
        {
            Value = rnd_mut.Next(GA.Generator_min_value, GA.Generator_max_value);
        }

        //Установка случайного значения гена
        public void Set_random_value(int min_value, int max_value, Random rnd)
        {
            Value = rnd.Next(min_value, max_value);
        }
    }
}
```

ПРИЛОЖЕНИЕ В СВИДЕТЕЛЬСТВА О РЕГИСТРАЦИИ ПРОГРАММЫ ДЛЯ ЭВМ

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2020663453

Программа генерации тестовых данных на основе модификации генетического алгоритма с использованием методов оценки сложности кода

Правообладатель: *ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ» (RU)*

Авторы: *Сердюков Константин Евгеньевич (RU), Авдеенко Татьяна Владимировна (RU)*

Заявка № 2020662849
Дата поступления 28 октября 2020 г.
Дата государственной регистрации в Реестре программ для ЭВМ *28 октября 2020 г.*

Руководитель Федеральной службы по интеллектуальной собственности

Г.П. Ивлиев

РОССИЙСКАЯ ФЕДЕРАЦИЯ

СВИДЕТЕЛЬСТВО

о государственной регистрации программы для ЭВМ

№ 2020666236

Программа генерации тестовых данных на основе расширенной функции приспособленности, учитывающей меру сложности кода и разнообразие особей в популяции

Правообладатель: ***ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ «НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ» (RU)***

Авторы: ***Сердюков Константин Евгеньевич (RU),***
Авдеенко Татьяна Владимировна (RU)

Заявка № **2020665707**
Дата поступления **07 декабря 2020 г.**
Дата государственной регистрации
в Реестре программ для ЭВМ ***07 декабря 2020 г.***

Руководитель Федеральной службы
по интеллектуальной собственности

Г.П. Ивлиев

ПРИЛОЖЕНИЕ Г АКТЫ О ВНЕДРЕНИИ РЕЗУЛЬТАТОВ РАБОТЫ

ООО «ДЕЖАВЮ»
630008, г. Новосибирск, ул. Сакко и Ванцетти, д. 77 офис 906
ИНН/КПП: 5405032793/540501001
ОГРН: 1185476098774
Тел: +7 (960) 780-14-56

исх. 2022/03-3
от 18 марта 2022 г.

АКТ
внедрения результатов диссертационной работы
Сердюкова Константина Евгеньевича

Настоящим актом подтверждается, что результаты диссертационной работы Сердюкова К.Е. по специальности 05.13.11 – Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей (технические науки) используется в ООО «Дежавю».
Алгоритмы генерации тестовых данных, предложенные Сердюковым К.Е., позволяют анализировать разработанные программные продукты и генерировать минимально необходимое количество наборов данных для проверки большей части кода автоматизировано, без необходимости подбирать данные вручную.
Внедрение в процесс разработки программного обеспечения ООО «Дежавю» алгоритмов генерации тестовых наборов с использованием эволюционных алгоритмов, разработанных Сердюковым К.Е., позволило обеспечить более качественное всестороннее тестирование с меньшими временными затратами, что привело к увеличению качеству разработанных программных продуктов.

Директор ООО «Дежавю» Мицкевич Д. А.

УТВЕРЖДАЮ

Проректор по научной работе

Новосибирского государственного

технического университета

_____ С.В. Брованов

«06» 04 2021 г.

АКТ

об использовании результатов диссертационной работы Сердюкова К.Е. в учебном процессе

Комиссия в составе:

председатель: Достовалов Д.Н., к.т.н., зав. каф. автоматизированных систем управления,

члены комиссии: Гриф М.Г., д.т.н., профессор, профессор каф. автоматизированных систем управления,

Мезенцев Ю.А., д.т.н., доцент, профессор каф. автоматизированных систем управления,

Муртазина М.Ш., к.т.н., доцент каф. автоматизированных систем управления.

составили настоящий акт о том, что научные результаты диссертационного исследования Сердюкова Константина Евгеньевича по специальности 05.13.11 «Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей» используются в учебном процессе на кафедре автоматизированных систем управления в курсах «Интеллектуальные информационные системы», «Интеллектуальные системы и технологии», «Программная инженерия» и «Методы оптимизации». Оригинальное программное обеспечение, реализованное в интегрированной среде разработки Microsoft Visual Studio 2019, успешно применяется в научно-исследовательских работах студентов, при выполнении квалификационных работ бакалавров и магистерских диссертаций.

Использование программного обеспечения на кафедре автоматизированных систем управления при обучении студентов по направлениям подготовки 09.03.03 Прикладная информатика», 09.03.01 «Информатика и вычислительная техника» и 09.04.03 «Прикладная информатика» позволяет познакомить студентов с современными научно-практическими разработками, применяемыми для проведения предметных исследований, предложить примеры использования эвристических алгоритмов для решения оптимизационных задач, в частности, для решения задачи автоматического подбора тестовых данных. Это способствует повышению качества учебного процесса.

Председатель комиссии: Д.Н. Достовалов

Члены комиссии: М.Г. Гриф

Ю.А. Мезенцев

М.Ш. Муртазина