УДК 681.3.06

*О.А. Слабоспицкая, О.Г. Мороз*

# ПОДХОД К СТРУКТУРНОМУ ТЕСТИРОВАНИЮ ПРОГРАММ С ИСПОЛЬЗОВАНИЕМ ГЕНЕТИЧЕСКОГО АЛГОРИТМА

Предложен генетический алгоритм для повышения эффективности тестирования программного обеспечения за счет выявления наиболее критических фрагментов путей в графе потока управления программы. Приведены результаты его апробации. Поскольку исчерпывающее тестирование зачастую невыполнимо уже для программ среднего размера, обычно проверяются только части программы – не обязательно наиболее подверженные ошибкам. Поэтому развит более избирательный подход к генерации тестовых данных, позволяющий выделять наиболее критические пути для их первоочередной проверки. Применение подхода может способствовать повышению эффективности тестирования.

## Введение

Сегодня проблема качества является главной для мировой индустрии программного обеспечения (ПО). Существует даже отдельная дисциплина (программная инженерия), которая занимается всеми аспектами повышения качества программных продуктов, начиная с подготовительных работ и заканчивая их изъятием из обращения. *Тестирование* – неотъемлемая составляющая программной инженерии, один из методов непрерывного улучшения качества разработанного ПО посредством выявления его дефектов, не обнаруженных ранее другими видами проверок [1–4].

Следует подчеркнуть, что тестирование любой программной системы является сложным, длительным и дорогостоящим видом деятельности, требующим около трети времени и почти половины общей стоимости разработки ПО [3–5]. Согласно исследованиям Национального института стандартов и технологии США [6], «национальные ежегодные затраты из-за неадекватной инфраструктуры тестирования ПО оцениваются в диапазоне от $22.2 до $59.5 млрд.» – приблизительно 0.6 процента американского валового национального продукта. Эта сумма не включает расходы из-за отказов программ в критических системах, что, к примеру, привелок нецелевому запуску ракет «Пэтриот» в системе обороны США в 1991 г. или к неудачной посадке спускаемого модуля на Марс в 1999 г., что обошлось потерей $165 млн.

Из вышесказанного следует, что существенно поднять качество ПО, значительно сократив сроки и стоимость разработки, можно, в частности, за счет повышения эффективности его тестирования.

В результате почти полувековых исследований для решения этой сложной проблемы предложены и частично автоматизированы различные способы генерации тестовых данных, как-то методы случайной, символической, динамической и т. п. генерации, объединяемые сегодня в группы подходов структурного и функционального тестирования [2–11]. Однако существенного прогресса в этом направлении пока что достичь не удалось, и генерация тестовых данных остается, в значительной степени, ручной деятельностью.

В последние годы интенсивно исследуются методы искусственного интеллекта, и прежде всего алгоритмы поиска (муравьиные алгоритмы, имитации отжига, генетические алгоритмы и т. п.), – в качестве лучшей альтернативы для разработки генераторов тестовых данных [7, 8, 12]. В частности, получены обнадеживающие результаты по использованию для этих генераторов эволюционных вычислений и генетических алгоритмов [10–15].

В работе представлены результаты исследований авторов по применению генетического алгоритма для повышения эффективности структурного тестирования ПО за счет выявления наиболее критических фрагментов путей в графе потоков управления программы.

Статья имеет следующую структуру: в разделе 1 рассмотрена проблема путевого тестирования в жизненном цикле ПО; раздел 2 посвящен описанию графа

© О.А. Слабопицкая, О.Г. Мороз, 2012

потоков управления; в разделе 3 описываются основные структуры генетического алгоритма; в разделе 4 предлагается новый алгоритм генерирования тестовых данных и в разделе 5 дается заключение.

## 1. Проблема путевого тестирования ПО

С появлением в 1995 году стандарта ISO/IEC 12207 [5], где все действия по созданию ПС систематизированы в виде отдельных процессов жизненного цикла, тестирование отнесено к *основным* процессам. Кроме того, отдельные задачи тестирования решаются в других процессах разработки. В частности, задачи планирования тестирования распределены по процессам: «Проектирование архитектуры системы», «Анализ требований к ПО», «Проектирование ПО». Автономное и интеграционное тестирование ПО выполняются в процессах «Построение ПО» и «Интеграция ПО», поскольку неразрывно с ними связаны. Таким образом, сегодня тестирование *интегрировано* с процессами разработки и рассматривается как непрерывная многоуровневая деятельность на протяжении всего жизненного цикла ПО независимо от его критичности . Для каждого уровня установлены цели (тестируемые характеристики), типы объектов (вся система, программные компоненты, модули) и методы тестирования [1–4].

Цели и объекты тестирования вместе определяют критерии формирования множества тестов: его полноты – «какой объем тестов достаточен для достижения установленной цели?» и состава – «какие тесты следует выбирать?». Первый вопрос касается определения критерия покрытия, а второй – критерия выбора типов тестов.

Методы динамического тестирования отвечают на оба вопроса с позиций обеспечения наиболее показательного поведения программы при ее выполнении на конечном множестве тестовых данных, Они различаются подходами к проектированию тестов – способами специального выбора тестовых данных из (вообще говоря) бесконечного входного пространства.

Традиционно выделяют две категории этих методов [1–3]:

– функциональное тестирование, при котором программа рассматривается как "черный ящик" и используется только информация о ее функциях без доступа к исходному коду;

– структурное тестирование, при котором используются только структура и код программы "белого ящика".

Для методов «белого ящика» одним из наиболее мощных критериев тестового покрытия является *покрытие решений* (decision coverage) [2–4]. В соответствии с ним, набор тестов должен обеспечить хотя бы однократное выполнение каждой ветви программы и каждого логического условия внутри нее.

Выполнение данного критерия требует постановки и решения *проблемы путевого тестирования* – выделения в графе потоков управления программы множества путей, содержащих все ее ветви, и формирования тестового набора, обеспечивающего максимально возможное (а в идеале – исчерпывающее) прохождение этих путей при заданных ограничениях на ресурсы тестирования.

Однако, в общем случае, все пути протестировать невозможно по нескольким причинам. Во-первых, программа может содержать бесконечное количество путей, например, если в ней есть циклы. Во-вторых, число путей в программе экспоненциально возрастает с ростом количества ветвей в ней, и многие из этих путей могут быть невыполнимыми. В-третьих, количество тестовых ситуаций слишком велико, так как каждый путь может быть охвачен несколькими тестовыми ситуациями. Поэтому проблема путевого тестирования сложных программ, разрабатываемых для решения практических задач автоматизации деловых процессов в различных предметных областях, может стать *NP*-сложной проблемой, для которой покрытие всех возможных путей либо невозможно в принципе, либо же требует нереалистичных вычислительных ресурсов.

Так как охватить все пути в программном обеспечении невозможно, проблема путевого тестирования сводится к эффективному выбору приоритетного

подмножества путей для выполнения и; тестовых данных, покрывающих эти пути.

Игнорирование проблемы путевого тестирования либо ее некорректное решение приводит к фатальным последствиям при традиционной организации разработки ПО согласно водопадной модели [1, 6], когда ошибки в ветвях низкоуровневых модулей, «пропущенных» при их автономном тестировании, вызывают трудно диагностируемые отказы ПО во время его приемочных испытаний. Достаточно серьезными эти последствия могут быть и при итеративных моделях разработки – от спиральной до RAD [1], – если модуль с «пропущенной» ветвью реализует базовую функциональность. Однако особую актуальность проблема путевого тестирования приобретает в связи с быстрым распространением, особенно в современных гибких методологиях, практики разработки, управляемой тестами – test driven development [1, 6]. Согласно этой практике, вначале создаются модульные тесты, описывающие требования к ПО, а потом – код модулей, для которых эти тесты должны выполняться. Таким образом, игнорирование критических ветвей в тестах создает предпосылки для появления ошибок при их реализации в соответствующем коде.

## 2. Взвешенный граф потока управления

Граф потока управления – ориентированный граф, в котором выделены две вершины, например start и end, и который удовлетворяет следующим условиям:

– в вершину start не входит ни одна дуга;

– из вершины end не выходит ни одна дуга;

– любая вершина достижима из вершины start;

– из любой вершины достижима вершина end.

В графе потока управления каждая вершина соответствует базовому *блоку* – прямолинейному участку кода, не содержащему ни операций передачи управления, ни точек, на которые управление передается из других частей программы. Имеется только два исключения: точка, на которую выполняется переход, является первой инструкцией в базовом блоке, и базовый блок завершается инструкцией перехода. Направленные дуги используются для представления инструкций перехода от блока к блоку. Также, в большинстве реализаций добавлены два специализированных блока: входной, через который управление передается графу (вершина start) и выходной, завершающий все пути в данном графе (вершина end) [16].

Пусть $N$ – конечное множеством вершин, а $E$ – конечное множеством дуг. Предполагаем, что множество $N$ содержит вершины start и end. Тогда граф потока управления является конечным множеством из вершин и дуг, которое обозначим как $G = (N, E)$. Дуга $(i, j)$ в $E$ соединяет две вершины $n_i$ и $n_j$ в $N$. Блоки и вершины маркированы так, чтобы блок $b_i$ соответствовал вершине $n_i$. Дуги $(i, j)$, соединяющиеся базовые блоки $b_i$ и $b_j$, подразумевают, что управление может передаваться от блока $b_i$ к блоку $b_j$.

Путем длины $k$ через граф потока управления называется последовательность $k$ дуг $(e_1, e_2, \ldots, e_k)$, удовлетворяющая условию: пусть $n_p$, $n_q$, $n_r$, и $n_s$ являются вершинами, принадлежащими $N$, и $0 < i < k$. Тогда если $e_i = (n_p, n_q)$ и $e_{i+1} = (n_r, n_s)$, то $n_q = n_r$.

Граф потока управления называется *взвешенным*, если существует некоторая функция (правило) $f$: $E \rightarrow R$ (функция на множестве дуг со значениями на множестве вещественных чисел). Сама функция $f$ называется *весовой*, а ее значение на той или иной дуге называется *весом* этой дуги. Любой подграф данного графа и любой путь в данном графе имеют свой вес: это сумма весов дуг, входящих в этот подграф или в этот путь.

Очевидно, что в зависимости от решаемой задачи и преследуемых целей весовая функция может иметь различный вид. В данной работе при взвешивании графа предлагается использовать правило равенства сумм весов всех дуг, входящих в произвольную вершину, и всех дуг, исходящих из нее. Суть этого правила состоит в следующем: предполагается, что каждая вершина графа (исключением является

вершина start) имеет свой входящий вес (сумма весов всех входящих в нее дуг). Поскольку вершина start не имеет явных входящих дуг, то ей присваивается фиктивный входящий вес *w*, который будем называть *опорным весом* графа потока управления. Опорный вес – регулируемый параметр. Обычно чем больше граф потока управления имеет вершин и дуг, тем большим следует брать его опорный вес.

Входящий вес каждой вершины распределяется среди всех исходящих из нее дуг. Исключением является вершина end, не имеющая явных исходящих дуг. Заметим, что единого правила распределения входящего веса произвольной вершины *j* по ее исходящим дугам нет. Однако в подавляющем большинстве случаев большие веса целесообразно назначать, так сказать, критическим дугам, т. е. дугам в составе путей, более подверженных ошибкам, ведущим к тяжелым последствиям.

Мы будем придерживаться следующего правила: $\beta$ процентов входящего веса вершины выделяется для дуг в последовательном пути (эти проценты затем делятся поровну между дугами, составляющими этот путь), а остальные $\theta = 100 - \beta$ процентов входящего веса резервируются для циклов и ветвей. Важно подчеркнуть, что $\beta < 50\%$. При этом, если некоторая вершина имеет только одну исходящюю дугу, то весь ее входящий вес присваивается этой дуге.

Тройка $(G, w, \beta)$, где $G$ – граф потока управления, является одним из возможных представлений взвешенного графа потока управления. При этом пара $(w, \beta)$ задает правило взвешивания графа.

## 3. Генетический алгоритм

Генетические алгоритмы (ГА) – адаптивные алгоритмы общего назначения, которые используют принципы генетики и эволюционной теории Дарвина для решения оптимизационных проблем. Особенно ГА эффективны в случае больших, сложных и слабо структурированных пространствах поиска, где классические методы и алгоритмы не подходят, неэффективны, трудоёмки или слишком ресурсозатратны [17]. Поскольку ГА используют биологические метафоры, то и применяющаяся терминология напоминает биологическую.

ГА начинает свою работу с создания, случайным образом, исходного множества “хромосом”, каждая из которых представляет собой одно из возможных решений конкретной проблемы. Хромосомы кодируются как последовательности конечной длины в некотором конечном алфавите, удобном для выполнения эволюционных операций. Зачастую результатом кодирования являются обычные битовые строки. Каждая буква, которая составляет хромосому, называется геном, а совокупность таких хромосом образует “популяцию”.

Качество хромосом популяции оценивается с помощью целевой функции. По результатам этой оценки определяется, какие хромосомы будут участвовать в последующем эволюционном процессе. Затем создается новое поколение хромосом. При этом используются *генетические операторы,* в результате выполнения которых новые хромосомы получаются путем комбинации свойств родителей.

Базовая структура ГА включает шаги, представленные на рис. 1.

*Репродукция* – это процесс копирования хромосом в соответствии с целевой функцией решаемой задачи. При этом хромосомы с «лучшим» значением целевой функции имеют большую вероятность включения в следующую генерацию. Оператор репродукции является искусственной версией натуральной селекции “выживания сильнейшего”. Репродукция определяет, как хромосомы будут отобраны, сколько и каких потомков каждая из них создаст. С ее помощью из текущей популяции создается новая, так называемая брачная популяция, из хромосом которой будут в последующем образованы брачные пары для скрещивания.

Оператор *кроссинговера* – метод обмена информацией между двумя хромосомами: он определяет процедуру для получения: потомства от двух родителей.

Кроссинговер моделирует передачу наследственности хромосомами и существенно зависит от типа задачи, так как он

должен создать только допустимых потомков. После его применения будем иметь две старые хромосомы и всегда получаем две новые хромосомы.

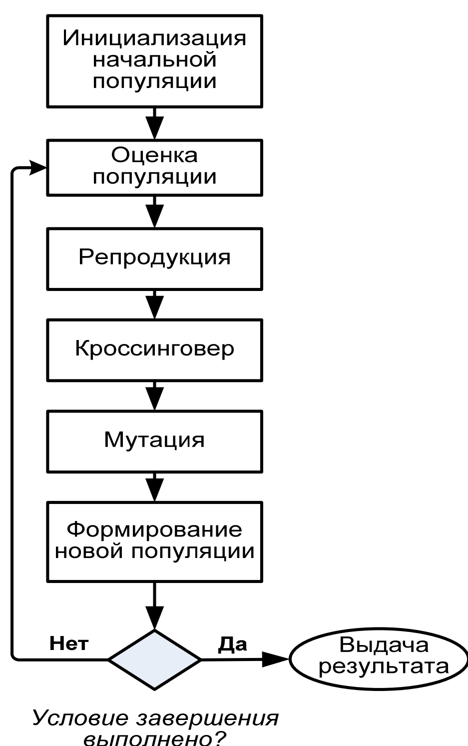


Рис. 1. Базовая структура генетического алгоритма

Сегодня предложено большое число различных операторов кроссинговера [11, 14, 17]. Наиболее распространенным, несмотря на свою простоту, является одноточечный кроссинговер. В соответствии с ним, две хромосомы, выбранные в качестве потенциальных родителей с помощью процесса репродукции и представленные в виде строк из “0” и “1”, обмениваются информацией путем обмена подстроками, находящимися справа от случайным образом выбранной точки разрыва.

Важно отметить, что каждая хромосома может образовать брачную пару с некоторой вероятностью $p_c$, которая называется вероятностью кроссинговера и является регулируемым параметром. Это происходит следующим образом: для каждого родителя генерируется случайное вещественное число $r$ в диапазоне [0, 1], и если $r < p_c$, то родитель отбирается для кроссинговера.

Оператор *мутации* изменяет значения одного или нескольких генов в хромосоме с целью повышения структурной изменчивости. Этот оператор является основным инструментом ГА для “выбивания” популяции из локального экстремума и ее защиты от преждевременной локализации в какой-либо конкретной области из всего поискового пространства [17]. В отличие от кроссинговера, мутация работает только с одной хромосомой за один раз. В большинстве случаев, мутация происходит сразу после кроссинговера, таким образом она фактически воздействует на потомков, напоминая естественный процесс.

Осуществляется мутация на побитовой основе. Каждый бит в любой хромосоме потомства имеет равные шансы мутации (изменения "0" на "1" или "1" на "0"), которая происходит в соответствии с вероятностью мутации $p_m$. Эта вероятность также является регулируемым параметром. Чтобы выполнить мутацию, для каждой хромосомы в потомстве и для каждого бита в хромосоме генерируют случайное вещественное число $r$ в диапазоне [0,1], и если $r < p_m$, то значение бита меняется на противоположное.

Если кроссинговер пытается создать лучшие хромосомы из наиболее пригодных, то мутация вносит разнообразие в популяцию, чтобы избежать возможности “застрять” в локальных оптимумах.

Действие и поведение ГА определяют различные параметры, среди которых наиболее важными являются:

– *размер популяции* $N$ – этот параметр определяет количество хромосом в популяции. Если $N$ слишком мал, то ГА может быстро сходиться, а если он слишком велик, то ГА может потратить впустую вычислительные ресурсы;

– *длина хромосомы* $L$ – определяет количество генов в каждой хромосоме. Это число зависит от выбранного представления;

– *количество поколений* $Ng$ – определяет число поколений, для которых ГА будет работать. Этот параметр часто используется в критериях завершения ГА;

– *вероятность кроссинговера* $p_c$ – вероятность обмена информацией между двумя родителями. Если значение $p_c$ слишком мало, то обмен информацией между

хромосомами с большим значением целевой функции может не состояться, следовательно уменьшается их способность произвести лучших потомков. С другой стороны, кроссинговер может также дать потомство с более низким значением целевой функции, и поэтому если значение $p_c$ слишком большое, то увеличивается вероятность попасть в локальный оптимум;

– *вероятность выполнения мутации* $p_m$ – вероятность мутации в данной хромосоме. Обычно $p_m < p_c$.

Мутация помогает уберечь популяцию от попадания в локальный оптимум, но с увеличением значения $p_m$ замедляется сходимость алгоритма.

## 4. Предлагаемый подход

Предлагаемый подход лучше всего продемонстрировать на конкретном примере. В качестве такого примера рассмотрим алгоритм Эвклида нахождения наибольшего общего делителя двух натуральных чисел, который часто используется при решении самых разнообразных задач. Его код и взвешенный граф потока управления показаны на рис. 2. Значения весов указаны возле каждой дуги.

Номера вершин соответствуют строкам кода. Поскольку строка 1 кода – условный оператор IF, то вершина 1 является узлом предиката и две исходящих из него дуги соответствуют двум возможным исходам этого оператора. Строка 6 является оператором WHILE и, соответственно, вершина 6 также имеет две исходящие дуги в построенном графе. К тому же, можно увидеть дугу от вершины 9 к вершине 6, отображающую петлю (цикл). На этом рисунке показано назначение веса дугам графа в соответствии с правилом (10, 20), описанным в разд. 2. Поскольку фрагмент кода небольшой, то опорный вес $w$ был взят равным 10 и затем распределен по различным дугам в зависимости от их важности. При этом для каждой вершины 20 % ее входящего веса отдается исходящим дугам в последовательном пути. Приоритетность пути при генерации тестовых данных основана на факте, что предикат, петля и вершины ветвления имеют предпочтение перед последовательными вершинами во время тестирования программного обеспечения.

Чтобы применить генетический алгоритм для конкретной проблемы, к примеру, для генерации тестовых случаев, мы должны определить следующие его элементы [17]:

– генетическое представление для потенциальных решений проблемы;

– целевую функцию;

– метод создания начальной популяции потенциальных решений;

– генетические операторы, которые изменяют состав потомков;

– значения различных параметров, используемых генетическим алгоритмом (размер популяции, вероятности применения генетических операторов и т. д.).

Наш алгоритм работает с графом потока управления. Путь, который будет выполнен программой, однозначно определяется значениями входных данных, т. е. парой натуральных чисел ($a, b$). Эта пара, с точки зрения ГА, и является хромосомой. Каждую хромосому кодируем в двоичном алфавите {0,1}. Так, к примеру, закодированная хромосома (15,4) представляет собой битовую строку вида: 11110100.

Качество каждой хромосомы оценивается целевой функцией вида:

$$F = \sum_{i=1}^{l} w_i ,$$

где $w_{\mathrm{i}}$ – вес, назначенный $i$-й дуге соответствующего хромосоме пути, $l$ – количество дуг, образующих путь. Например, хромосоме (15,4) соответствует путь 0-1-2-3-4-5-6-7-8-9-6-7-8-9-6-10-11-12 и поэтому значение целевой функции этой хромосомы равно 108.

Начальная ($X_0$) и последующие популяции ($X_i$) представляют собой множества из $N$ случайно сгенерированных двоичных последовательностей, каждая из которых является закодированной хромосомой. Однако, для удобства изложения основных идей работы, мы обычно будем обозначать хромосому в явном виде, т. е. как множество пар тестовых данных, сгенерированных с помощью датчика случайных чисел, т. е.

$$X_i = \{(a_{i1},b_{i1}), (a_{i2},b_{i2}),\ldots,(a_{iN},b_{iN})\},$$

```
0       gcd (int a, int b)
        {
        int r;
1       if (b>a) {
2       r = a;
3       a = b;
4       b=r;
        }
5       r = a%b;
6       while(r!=0)
        {
7       a = b;
8       b=r;
9       r = a%b;
10      }
11      return b;
12      }
```

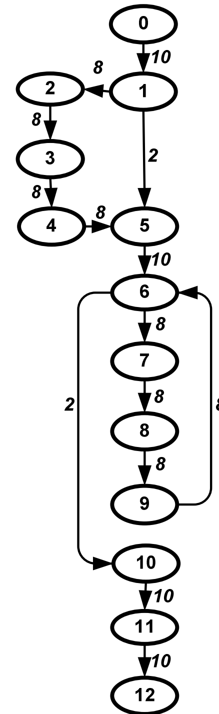


Рис. 2. Код пограммы и еевзвешенный граф потока управления

где $N$ – размер популяции, $i$ = 0,1,… – ее номер, $a_{ik}$ и $b_{ik}$ – натуральные числа.

Отбор лучших родителей популяции для участия в последующем воспроизводстве осуществляется с помощью пропорциональной селекции.

При этом вероятность $p_j$ выбора хромосомы $j$ находится по формуле:

$$p_j = F_j / \sum_{i=1}^{N} F_i ,$$

где $N$ – начальный размер популяции.

Кумулятивная вероятность $k$-й хромосомы $C_k$ рассчитывается следующим образом: $C_k = \sum_{j=1}^{k} p_j$ .

В результате применения одноточечного кроссинговера с точкой разрыва между четвертым и пятым битами входной битовой строки каждая пара отобранных из брачной популяции хромосом с вероятностью $p_c$ = 0,9 производит две новые хромосомы путем обмена правыми от точки разрыва частями.

После получения всех хромосом-потомков, каждая из них подвергается мутации. При этом для каждого бита хромосомы генерируется случайное число, и если оно меньше $p_m$ = 0.3, то значение соответствующего бита меняется на противоположное.

На завершающем этапе каждого прогона алгоритма из хромосом текущей популяции, т. е. родителей, и их потомков одним из известных методов, например методом вытеснения, формируем новую популяцию. Если условие завершения алгоритма не выполняется, то начинаем следующий прогон алгоритма уже с новой популяцией.

Результаты решения задачи представлены в табл. 1–7. Обозначения, использующиеся в этих таблицах, следующие:

$X_i$ – $i$-я популяция хромосом;

$F_{ik} = F(\gamma_{ik})$ – значение целевой функции $k$-й хромосомы $i$-й популяции, где $\gamma_{ik}$ обозначает $k$-ю хромосому $i$-й популяции, т. е. тестовые данные ($a_{ik}$, $b_{ik}$);

$P_{ik}$ – вероятность случайного выбора $k$-й хромосомы $i$-й популяции т. е.

$$P_{ik} = \frac{F(\gamma_{ik})}{\sum_{k=1}^{n} F(\gamma_{ik})},$$

$C_{ik}$ – кумулятивная вероятность $k$-й хромосомы $i$-й популяции;

$R_{ik}$ – сгенерированное случайное число, необходимое для реализации опера-

тора репродукции в $i$-й популяции, $k = 1,\ldots,N$;

$N_{ik}$ – номер хромосомы $i$-й популяции, кумулятивная вероятность которой превышает $R_{ik}$;

БП – брачная популяция. В этом столбце указано, сколько раз каждая хромосома появляется в столбце $N_{ik}$.

Начальная популяция:

$$X_0 = \{(12, 8), (2, 3), (6, 2), (15, 4)\}$$

В табл. 1 – 7 приведены результаты итераций выполнения алгоритма. Значения целевой функции текущей популяции приведены в столбце 3. В столбце 5 даны значения кумулятивной вероятности. Случайные числа, необходимы для реализации оператора репродукции, приведены в столбце 6. В столбце 8 показано, какое количество хромосом разного вида текущей популяции содержится в брачной популяции.

Результаты итераций описываются таблицами следующим образом:

Итерация 1: табл. 1 и 2 .

Итерация 2: табл. 3 и 4 .

Итерация 3: табл. 5 и 6 .

Конечный результат: табл. 7.

Результатом применения рассмотренного алгоритма является набор тестовых данных, состоящий из пар (4, 3) и (7, 12). Как видно из табл. 7, он обеспечивает полное покрытие путей программы.

Таблица 1

| N п/п | $X_0$ | $F_{0k}$ | $P_{0k}$ | $C_{0k}$ | $R_{0k}$ | $N_0$ | БП |
|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | (12,8) | 76 | 0.228 | 0.228 | 0.934 | 4 | 0 |
| 2 | (2,3) | 106 | 0.317 | 0.545 | 0.474 | 2 | 2 |
| 3 | (6,.2) | 44 | 0.132 | 0.677 | 0.374 | 2 | 1 |
| 4 | (15,4) | 108 | 0.323 | 1.000 | 0.618 | 3 | 1 |

Таблица 2

| N п/п | $N_0$ | БП | Кроссинговер | Мутация |
|---|---|---|---|---|
| 1 | 4 | (15,4)<br>11110100 | (15,4)<br>11110100 | (15,5)<br>11110101 |
| 2 | 2 | (2,3)<br>00100011 | (2,3)<br>00100011 | (3,3)<br>00110011 |
| 3 | 2 | (2,3)<br>00100011 | (2,2)<br>00100010 | (2,2)<br>00100010 |
| 4 | 3 | (6,2)<br>01100010 | (6,3)<br>01100011 | (10,3)<br>10100011 |

Таблица 3

| N п/п | $X_1$ | $F_{1k}$ | $P_{1k}$ | $C_{1k}$ | $R_{1k}$ | $N_1$ | БП |
|---|---|---|---|---|---|---|---|
| 1 | (15,5) | 44 | 0.212 | 0.212 | 0.217 | 2 | 0 |
| 2 | (3,3) | 44 | 0.212 | 0.424 | 0.999 | 4 | 1 |
| 3 | (2,2) | 44 | 0.212 | 0.636 | 0.979 | 4 | 1 |
| 4 | (10,3) | 76 | 0.364 | 1.000 | 0.533 | 3 | 2 |

Таблица 4

| N п/п | $N_s$ | БП | Кроссинговер | Мутация |
|---|---|---|---|---|
| 1 | 2 | (3,3)<br>00110011 | (3,2)<br>00110010 | (5,10)<br>01011010 |
| 2 | 4 | (10,3)<br>10100011 | (10,3)<br>10100011 | (9,10)<br>10001010 |
| 3 | 4 | (10,3)<br>10100011 | (10,3)<br>10100011 | (11,6)<br>10110110 |
| 4 | 3 | (2,2)<br>00100010 | (2,3)<br>00100011 | (2,2)<br>00100010 |

Таблица 5

| N п/п | $X_2$ | $F_{2k}$ | $P_{2k}$ | $C_{2k}$ | $R_{2k}$ | $N_2$ | БП |
|---|---|---|---|---|---|---|---|
| 1. | (5,10) | 74 | 0.223 | 0.223 | 0.934 | 4 | 0 |
| 2. | (9,10) | 106 | 0.319 | 0.542 | 0.474 | 2 | 2 |
| 3. | (11,6) | 108 | 0.325 | 0.867 | 0.374 | 2 | 1 |
| 4. | (2,2) | 44 | 0.133 | 1.000 | 0.618 | 3 | 1 |

Таблица 6

| N п/п | $N_s$ | БП | Кроссинговер | Мутация |
|---|---|---|---|---|
| 1 | 4 | (2,2)<br>00100010 | (2,2)<br>00100010 | (4,3)<br>01000011 |
| 2 | 2 | (9,10)<br>10011010 | (9,10)<br>10011010 | (7,12)<br>01111100 |
| 3 | 2 | (9,10)<br>10011010 | (9,10)<br>10011010 | (13,10)<br>11011010 |
| 4 | 3 | (11,6)<br>10110110 | (11,6)<br>10110110 | (2,6)<br>00100110 |

Таблица 7

| $N$ п/п | $X_3$ | $F_{3k}$ | $P_{3k}$ | $C_{3k}$ | $R_{3k}$ | $N_3$ | БП |
|---|---|---|---|---|---|---|---|
| 1 | (4,3) | 76 | 0.178 | 0.178 | 0.098 | 1 | 1 |
| 2 | (7,12) | 170 | 0.399 | 0.577 | 0.275 | 2 | 3 |
| 3 | (13,10) | 106 | 0.249 | 0.826 | 0.325 | 2 | 0 |
| 4 | (2,6) | 74 | 0.174 | 1.000 | 0.487 | 2 | 0 |

## Заключение

Генетические алгоритмы часто используются для решения проблем оптимизации, в которых развитие популяции является механизмом поиска удовлетворительного решения при ряде ограничений.

Зачастую они превосходят по эффективности исчерпывающий поиск и методы локального поиска.

В работе предложен генетический алгоритм для повышения эффективности путевого тестирования, основанный на выявлении и анализе наиболее критических путей графа потока управления.

Представлены результаты эксперимента, сравнивающего случайную генерацию тестовых данных с новым подходом, использующим генетический поиск.

Применение подхода может способствовать сокращению трудозатрат на проведение тестирования и его стоимости. Поскольку эксперименты были проведены для относительно небольших программ, то дальнейшее усовершенствование предложенного подхода требует его апробации для программных продуктов различного размера и назначения.

1. *Андон Ф.И., Коваль Г.И., Коротун Т.М., Лаврищева Е.М., Суслов В.Ю.* Основы инженерии качества программных систем. 2-е издание. – Киев.: Акадкмпериодика, 2007. – 672 с.
2. *Липаев В.В.* Тестирование программ. – М.: Радио и связь, 1986. – 296 с.
3. *Beizer B.* Software Testing Techniques. 2nd. Ed. Van Nostrand Reinhold, 1990. – 549 c.
4. *Mathur A.P.* Foundations of Software Testing: Fundamental Algorithms and Techniques, 1st edition Pearson Education.– 2008. – С. 689.
5. ISO/IEC 12207: 1995. Information technologies. Software life cycle processes. – 61 p.
6. *Pfleeger S.L.* Software Engineering: Theory and Practice. 2nd Edition, Prentice-Hall, 2001. – 659 p.
7. *Mansour N., Salame M.* Data Generation for Path Testing , Software Quality J. – 2004.– 12.– P. 121–136.
8. *Wegener J., Baresel A., Sthamer H.* Suitability of Evolutionary Algorithms for Evolutionary Testing // In Proceedings of the 26th Annual International Computer Software and Applications Conference, Oxford, England, August 26– 29, 2002.
9. *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), University of South Florida, March 25–26, 2004.– p. 261–262.
10. *Korel B.* Automated software test data generation. IEEE Transactions on Software Engineering, 16(8), August 1990.
11. *Jones B.F., Sthamer H.H., Eyres D.E.* Automatic structural testing using genetic algorithms. Software Engineering J., September, 1996. – p. 299–306.
12. *Harman M., Jones B.* Search-based Software engineering // Information and software technology. – 2001, 43(14). – p. 833–839.
13. *Last M., Eyal S., Kandel A.* Effective Black-Box Testing with Genetic Algorithms // Lecture notes in computer science. – 2006. – P. 134–148.
14. *Alander J., Mantere T., Turunen P.* Genetic Algorithm Based Software Testing. – http://cite-seer.ist.psu.edu/40769.html.
15. *Berndt D.J., Fisher J., Johnson L., Pinglikar J., Watkins A.* Breeding Software Test Cases with Genetic Algorithms // In Proceedings of the Thirty-Sixth Hawaii International Conference on System Sciences (HICSS-36), Hawaii, January 2003.
16. *Allen F.E.* Control flow analysis/ACM SIGPLAN Notices – Proceedings of a symposium on Compiler optimization. –1970. – Vol.5, Issue 7. – p. 1–19.
17. *Гладков Л.А., Курейчик В.В., Курейчик В.М.* Генетические алгоритмы. – М.: ФИЗМАТЛИТ, 2006. – 320 с.

Получено 26.07.2011

***Об авторах:***

*Слабоспицкая Ольга Александровна*,
кандидат физико-математических наук,
старший научный сотрудник,

*Мороз Ольга Григорьевна*,
инженер-программист.

***Место работы авторов:***

Институт программных систем
НАН Украины,
03187, Киев-187,
проспект Академика Глушкова, 40.
Тел.: (044) 526 4579
e-mail: ols07@mail/ru