Исследование способов оценки сложности программного кода при генерации входных тестовых данных

К.Е. Сердюков[1], Т.В. Авдеенко[1]

[1]Новосибирский государственный технический университет, пр-т К. Маркса 20, Новосибирск, Россия, 630073

Аннотация. В данной статье предлагается сравнение методов определения сложности кода при генерации наборов данных для тестирования программного обеспечения. В статье предлагаются результаты исследования по оценке одного пути программного кода, работа ещё не окончена, в дальнейшем будет расширена для подбора данных для тестирования множества путей. Для решения задачи генерации наборов данных, предлагается использовать генетический алгоритм с различными способами определения сложности программного кода. Предлагается новый метод определения сложности кода на основе изменения весов вложенных операций. В статье приводятся результаты и сравнение сгенерированных входных тестовых данных на предмет прохода по критическому пути. Для каждой рассмотренной в статье метрике представлены выводы для выявления особенностей в зависимости от подобранных данных.

1. Введение

Программная инженерия является комплексным, систематическим подходом к разработке и сопровождению программного обеспечения. При разработке программ чаще всего выделяют следующие этапы – анализ, проектирование, программирование и тестирование. На этапе анализа определяются требования к программному обеспечения и производится документирование. На этапе проектирования детализируется внешний вид программы, определяется её внутренний функционал, разрабатывается структура продукта и вводятся требования для последующего тестирования. Написание исходного кода программы на одном из языков программирования производится на этапе программирования.

Одним из наиболее важных шагов при разработке программных продуктов является тестирование. Важными целями тестирования являются соответствие разработанной программы заданным требованиям, соблюдение логики в процессах обработки данных и получение верных конечных результатов. Поэтому для тестирования очень важно сгенерировать входные данные, на основе которых программа будет проверяться на наличие ошибок и соответствие заданным требованиям. Для оценки качества входных данных используется показатель покрытия кода, то есть какой процент всей программы предложенные тестовые наборы могут «покрыть». Определяется отношением тестируемых операций ко всему количеству операций в коде.

Некоторые процессы тестирования программного кода совершенствуются достаточно медленно. Разработка большинства видов сценариев тестирования чаще всего происходит вручную, без использования каких-либо систем автоматизации. Из-за этого процесс тестирования становится невероятно сложным и затратным как по времени, так и по финансам,

если подходить к нему со всей серьёзностью. Для тестирования некоторых программ может уходить до 50% всех временных затрат.

Одной из основных целей тестирования является создание такого тестового набора, который обеспечивал бы достаточный уровень качества конечного продукта за счёт проверки большинства различных путей программного кода, т.е. обеспечивал бы его максимальное покрытие. Тем не менее, сама задача поиска многих путей состоит из нескольких подзадач, решение которых необходимо для нахождения качественного набора данных. Одной из локальных задач, решаемых для поиска тестового набора, является определение одного, наиболее сложного пути кода.

По большей части валидация и верификация программных продуктов плохо поддаётся оптимизации. Особенно сложно автоматизировать генерацию тестовых данных, которая по больше части делается вручную.

Тем не менее, существует множество исследований с использованием нестандартных алгоритмов для решения проблемы автоматизации. Для примера, в статье [1] предлагается использовать алгоритм, основанный на ограничениях, для системы Морта (анг. Morth), использующего тестирование на наличие ошибок для нахождения входных тестовых данных. Тестовые данные подбираются таким образом, чтобы определить наличие или отсутствие определённых ошибок.

Достаточно часто для решения данной проблемы тем или иным образом используются генетические алгоритмы. В статье [2] сравниваются разные методы генерации тестовых данных, включая генетические алгоритмы, метод случайного поиска и другие эвристические методы.

В статье [3] для решения проблемы предлагается использовать логическое программирование в ограничениях (анг. Constraint Logic Programming) и символьное исполнение. В [4] используются правила обработки ограничений (анг. Constraint Handling Rules, CHR) для помощи в ручной верификации проблемных мест в программе.

Некоторые исследователи применяют эвристические методы для автоматизации процесса тестирования при помощи диаграммы потоков данных (анг. Data-flow diagram). Исследования методов автоматизации с использованием данной диаграммы были проведены в статьях [5, 6, 7, 8]. В статье [5] предлагается дополнительно использовать генетические алгоритмы для генерации новых входных наборов тестовых данных на основе ранее использованных.

В статьях [9, [10] предлагается использовать гибридные методы генерации тестовых данных. В [9] применяется подход, объединяющий стратегии случайного поиска (анг. Random Strategy, RS), динамического символьного исполнения (анг. Dynamic Symbolic Execution, DSE) и стратегий на основе поиска (анг. Search-Based Strategy, SBS). В статье [10] предлагается теоретическое описание метода поиска с применением генетического алгоритма. Рассматриваются подходы поиска локальных и глобальных экстремумов на реальных программах. Предлагается гибридный подход генерации тестовых данных – меметический алгоритм (анг. Memetic Algorithm).

Подход в [11] для генерации тестовых данных используется алгоритм гибридного интеллектуального поиска. В предложенном подходе особое место уделяется методам ветвей и границ и поиска экстремумов (анг. Hill Climbing) для улучшения интеллектуального поиска.

Так же существуют исследования с использованием машинного обучения, например в статье [12]. В ней предлагается метод с использованием нейронной сети и настраиваемой пользователем кластеризации входных данных для последовательного обучения.

Для генерации тестовых данных может быть так же применён подход поиска новизны (анг. Novelty Search, NS). В статье [13] предлагается использовать данный подход для просмотра больших пространств входных данных и сравнивается с подходами на основе генетического алгоритма.

Так же исследуются возможности генерации тестовых данных для тестирования веб сервисов, например в спецификации WDSL [14].

Для удобства генерации тестовых данных применяются так же UML диаграммы. [15, 16]. В статьях предлагается использовать генетические алгоритмы для генерации триггеров для UML

диаграмм, которые позволят отыскать критический путь в программе. В статье [17] предлагается улучшенный метод на основе генетического алгоритма для подбора тестовых данных для множества параллельных путей в UML диаграммах.

Кроме UML диаграмм, программа может быть отображена в виде метода дерева классификаций (анг. Classification-Tree Method), разработанным Грочатманном (анг. Grochtmann) и Гриммом (анг. Grimm) [18]. В статье [19] рассматриваются проблема построения деревьев и предлагается интегрированный алгоритм дерева классификаций, а в [20] исследуется разработанный прототип ADDICT (сокр. AutomateD test Data generation using the Integrated Classification-Tree methodology) для интегрированного подхода.

В данной статье предлагается сравнение разных методов оценки сложности кода для генерации тестовых данных. Статья структурирована следующим образом. В пункте 2 вводится терминология и даётся основная информация по генетическому алгоритму. Во третьем пункте ставится задача к решению и вводится один из методов оценки сложности кода. В пункте 4 предлагаются результаты работы алгоритма генерации входных данных с использованием введённого метода оценки кода. В 5 проводится сравнение разных методов оценки кода.

2. Генетические алгоритмы

Формально, генетический алгоритм не является оптимизационным методом, по крайне мере в понимании классических методов оптимизации. Его цель заключается не в нахождении оптимального и самого лучшего решения, а в нахождении достаточно близкого к нему. Поэтому данный алгоритм не рекомендуют применять, если уже существуют быстрые и хорошо проработанные методы оптимизации. Но при этом, генетический алгоритм отлично показывают себя в решении не стандартизированных задач, задач с неполными данными или для которых невозможно применение оптимизационных методов из-за сложности реализации или длительности выполнения [21,[22].

Генетический алгоритм считается выполненным, если пройдено определённое число итераций (количество итераций желательно ограничивать, так как генетический алгоритм работает на основе проб и ошибок, что является достаточно длительным процессом), либо, если было получено удовлетворительное значение функции приспособленности. Как правило, генетический алгоритм решает задачи максимизации или минимизации и адекватность каждого решения (хромосомы) оценивается с помощью функции приспособленности.

Генетический алгоритм работает последующему принципу:

- Происходит инициализация. Вводится функция приспособленности. Формируется начальная популяция. В классической теории начальная популяции формируется случайным заполнением каждого гена в хромосомах. Но для увеличения скорости сходимости решения, начальная популяция может быть задана определённым образом, либо заранее проанализированы случайные значения для исключения определённо не подходящих генов.
- Оценка популяции. Каждая из хромосом оценивается функцией приспособленности. На основе заданных требований, хромосомы получают точное значение, насколько хорошо они соответствует решаемой проблеме.
- Селекция или отбор. После того, как каждая из хромосом получила собственное значение, происходит отбор наилучших хромосом. Селекция может производиться разными методами, например, из отсортированных по порядку выбираются первые n хромосом, либо будут выбираться только наиболее подходящие, но не меньше n и т.д.
- Скрещивание.[23]. Первое существенно отличие от обычных методов. После селекции и выбора подходящих для решения проблемы хромосом, они скрещиваются между собой. Случайные хромосомы из всех «избранных» в случайном порядке образовывают новые хромосомы. Скрещивание происходит на основе выбора определённой позиции в двух хромосомах и замена частей друг друга. После того, как заполнится необходимое число хромосом для создания популяции, алгоритм переходит к следующему шагу.
- Мутация. [24]. Также шаг, характерный только для ГА. В случайном порядке случайный ген может поменять значения на случайное. Основной смысл в мутация такой же, как и

в биологии – привнести генетическое разнообразие в популяцию. Главная цель мутаций состоит в получении решений, которые не могли бы получиться с имеющимися генами. Это позволит, во-первых, избежать попадания в локальные экстремумы, так как мутация может позволить перевести алгоритм в совершенно другую ветвь и во-вторых, «разбавить» популяцию, чтобы избежать ситуации, когда во всей популяции будут только одинаковые хромосомы, которые не будут вообще двигаться к решению.

После того, как проведены все шаги, оценивается, достигла ли популяция желаемой точности решений или пришла к ограничению количества популяций, и если да, то алгоритм прекращает работу. Иначе цикл уже с новой популяцией повторяется, пока условия не будут достигнуты.

3. Постановка задачи

Использование генетических алгоритмов в процессе тестирования позволяют обеспечить нахождение наиболее сложных частей программы, в которых риски из-за допущения ошибок наиболее велики. Оценивание происходит за счёт использования функции приспособленности, в качестве параметров которой выступают веса каждой проходимой операции. Определение весов, т.е. сложности программного кода, происходит за счёт различных метрик, применяемых в зависимости от требований к наборам.

Задача по поиску входных тестовых данных состоит из трёх подзадач:

1) Поиск входных данных для прохода по одному наиболее сложному пути кода. Сложность определяется выбранной метрикой для оценки кода;
2) Исключение или снижение весов операций на пути, для которого были подобраны данные, из расчёта функции приспособленности для других путей;
3) Генерация входных тестовых данных для множества путей программного кода.

Ограничение на количество наборов входных данных устанавливается после этапа разработки и позволит сконцентрироваться на определённых путях.

Весь алгоритм выполняется циклически – запускается процедура поиска входных данных для одной ветви, после чего операции в этой ветви исключаются из дальнейших вычислений и снова запускается поиск данных для одной ветви.

В качестве одного из способов определения сложности кода предлагается алгоритм, который работает следующим образом:

- Первой операции назначается вес, например, в 100 единиц.
- Каждой последующей операции так же назначается вес – если нет никаких условий или циклов, вес берётся в соответствии с предыдущей операцией.
- Условия разделяют вес в соответствии с правилом – если условие содержит только одну ветвь (только if…), то вес каждой операции снижается на 80%. Если условие разделяется на несколько ветвей (if…else…), то вес делится на равнозначные части – для двух ветвях 50%/50%, для трёх 33%/33%/33% и т.д.
- Веса операций в цикле остаются, но также могут умножаться на определённых вес, если необходимо.
- Все вложенные ограничения суммируются, например, для двух вложенных условий вес операций будет равен 80% * 80% = 64%

Назначенные веса можно использовать для того, чтобы разрабатывать тестовые варианты при помощи генетических алгоритмов, то есть для оценки того, как много рассчитываемого веса приходится на тут или иную ветвь при определённых значениях входных параметров.

Для удобства представления введём следующие обозначения:

X – наборы данных;

F(X) – значение функции приспособленности для каждого набора данных в зависимости от рассчитанных значений весов.

Задача заключается в максимизации целевой функции, т.е. $F(X) \rightarrow max$.

4. Исследование результатов работы алгоритма

В соответствии с ранее предложенным вариантом оценки сложности программного кода, данный метод дорабатывается для лучшего соответствия реальным требованиям. Вес считаются в соответствии с работоспособностью программы, другими словами, чем больше итераций выполнит программа, тем больший вес будет иметь изначальный тестовый вариант.

Первая популяция формируется случайными значениями. В каждой популяции содержится по 100 хромосом. Общее число популяция также равняется 100. Благодаря этому сформируется достаточное число различных вариантов и выберутся наилучшие из них.

В таблице 1 представлены итоги тестирования.

Таблица 1. Сравнение результатов запуска метода.

Популяция	Тест 1	Тест 2	Тест 3	Тест 4
0	1: 78, 23, 35	1: 97, 3, 6	1: 92, 97, 28	1: 15, 67, 26
	2: 62, 36, 95	2: 82, 77, 64	2: 38, 66, 52	2: 32, 27, 83
	3: 52, 35, 27	3: 24, 47, 57	3: 63, 76, 64	3: 37, 52, 64
	4: 17, 77, 73	4: 90, 13, 82	4: 7, 24, 56	4: 70, 49, 64
	5: 75, 9, 96	5: 81, 69, 24	5: 57, 48, 8	5: 67, 29, 94
20	1: 95, 64, 54	1: 97, 80, 4	1: 99, 13, 10	1: 99, 71, 45
	2: 95, 64, 29	2: 97, 80, 53	2: 99, 13, 11	2: 99, 71, 15
	3: 95, 64, 54	3: 97, 80, 28	3: 99, 13, 11	3: 99, 71, 3
50	1: 95, 64, 54	1: 97, 80, 29	1: 99, 13, 10	1: 99, 71, 60
	2: 95, 64, 29	2: 97, 80, 4	2: 99, 13, 11	2: 99, 71, 3
	3: 95, 64, 54	3: 97, 80, 53	3: 99, 13, 11	3: 99, 71, 3
Итоговая	1: 95, 64, 54	1: 97, 80, 4	1: 99, 13, 10	1: 99, 71, 60
(100)	2: 95, 64, 29	2: 97, 80, 29	2: 99, 13, 11	2: 99, 71, 45

В каждом из тестов сформировались как минимум два различных варианта данных, при которых рассматриваемый программный код отработает больше всего раз, а значит и большее число раз пройдёт по различным путям. Кроме того, можно увидеть определённые закономерности в результатах – первое значение всегда максимально (случайные значения ограничивались значением 100), второе значение меньше, чем первое, но больше 3-го.

5. Сравнение методов оценки сложности программного кода

Для исследования проведено несколько тестов работы алгоритма с четырьмя различными метриками – модифицированной метрикой, логика которой была описана в п. 3, метрики SLOC по оценке количества строк кода, метрики ABC и метрики Джилба.

Метрика SLOC (анг. полн. Source Lines of Code) определяется количеством строк кода. В данной метрике учитывается только общее количество строк кода в программе, что делает её самой простой для понимания. В данном случае под количеством строк понимается количество команд, а не физическое количество строк.

ABC-метрика, или метрика Фитцпатрика (анг. Fitzpatrick) является мерой, которая определяется на основе трёх различных показателей ABC = $<n_a,n_b,n_c>$. Первый показатель n_a (от анг. Assignment) выделен под строки кода, которые отвечают за назначение переменным определённого значения, например, int number = 1. Показатель n_b (от анг. Branch) отвечает за использование функций или процедур, то есть операндов, которые работают вне видимости текущего программного кода. Показатель n_c (от анг. Condition) подсчитывает количество логических операндов, таких как условия и циклы. Значение метрики высчитывается как квадратный корень из суммы квадратов значений n_a, n_b и n_c.

$$F = \sqrt{n_a{}^2 + n_b{}^2 + n_c{}^2} \quad (1)$$

Примечательно, что одна строка кода может учитываться в разных показателях, например, при назначении переменной значения определённой функции (double number = Math.Pow(2, 3), присваивание переменной number значения 2 в степени 3 учитывается как в n_a, так и в n_b). К

недостаткам данной метрики относят возможное возвращение нулевого значения в некоторых частях кода.

Метрика Джилба (анг. Jilb) направлена на определение сложности программного кода в зависимости от его насыщенности условными операндами. Данная метрика полезна для определения трудоёмкости программного кода, как для написания, так и для его понимания:

$$F = cl/n, \quad (2)$$

где cl – количество условных операндов, n – общее количество строк кода.

Для тестирования используется код со множество различных путей, где один является критическим, то есть на нём находится наибольшее число операций. Подбор входных данных для данного пути будет являться решением подзадачи и по этим данным можно будет определить, насколько точно подбираются данные. Выход на критический путь будет в том случае, если из подобранных данных 1-е и 3-е значения больше 50 и 1 значение меньше 3.

Для генерации входных тестовых данных используются следующие настройки генетического алгоритма:

- Количество поколений – 100
- Количество популяций в одном поколении – 100
- Диапазон принимаемых значений данных – (0, 100)

5.1. Результаты с использованием предложенной в статье метрики

Алгоритм с данной метрикой подбирает данные с приоритетом операций более высокого уровня. В результате (99 поколение) были получены два набора данных – (70, 9, 78) и (75, 67, 82). Оба набора проходят по самому длинному пути кода, что является решением подзадачи. В таблице 2 представлены первые 10 вариантов в каждом из поколений.

Таблица 2. Результаты работы предложенной метрики.

Вариант\Поколение	0	1	99
	Модифицированная метрика		
1	(70, 9, 78) = 164100	(75, 67, 82) = 164100	(70, 9, 78) = 164100
2	(75, 67, 82) = 164100	(61, 29, 94) = 164100	(70, 9, 78) = 164100
3	(61, 29, 94) = 164100	(63, 52, 87) = 164100	(75, 67, 82) = 164100
4	(63, 52, 87) = 164100	(63, 49, 83) = 164100	(75, 67, 82) = 164100
5	(63, 49, 83) = 164100	(70, 9, 78) = 164100	(75, 67, 82) = 164100
6	(5, 68, 90) = 96382	(63, 52, 87) = 164100	(75, 67, 82) = 164100
7	(60, 37, 3) = 32500	(70, 9, 78) = 164100	(75, 67, 82) = 164100
8	(12, 80, 49) = 16000	(70, 9, 78) = 164100	(70, 9, 78) = 164100
9	(47, 12, 17) = 16000	(70, 9, 78) = 164100	(70, 9, 78) = 164100
10	(53, 35, 76) = 16000	(61, 29, 94) = 164100	(75, 67, 82) = 164100

Можно увидеть, что алгоритм работает достаточно эффективно и уже в первом поколении подобраны данные для критического пути.

5.2. Результаты с использованием метрики SLOC

Данная метрика является самой простой с точки зрения реализации, в ней учитывается только общее количество строк кода. Результаты работы представлены в таблице 3. Алгоритм с данной метрикой подобрал 3 набора - (63, 72, 91), (68, 50, 94) и (80, 70, 88). Все три удовлетворяют условиям для прохода по критическому пути.

Как и с предыдущей метрикой, алгоритм ещё на первом поколении подобрал подходящие данные.

5.3. Результаты с использованием метрики ABC

Данная метрика учитывает больше вариантов значений, таких как присваивание значений переменным, логических проверок и вызовов функций. Алгоритм с метрикой ABC подобрал 2

варианта входных данных, которые проходят по критическому пути – (69, 46, 78) и (77, 36, 98). Остальные варианты представлены в таблице 4.

Таблица 3. Результаты работы алгоритм с метрикой SLOC.

Метрика SLOC			
Вариант\Поколение	0	1	99
1	(64, 14, 96) = 6411	(68, 50, 94) = 6411	(63, 72, 91) = 6411
2	(68, 50, 94) = 6411	(80, 70, 88) = 6411	(68, 50, 94) = 6411
3	(80, 70, 88) = 6411	(65, 81, 89) = 6411	(68, 50, 94) = 6411
4	(65, 81, 89) = 6411	(63, 72, 91) = 6411	(63, 72, 91) = 6411
5	(63, 72, 91) = 6411	(74, 83, 76) = 6411	(80, 70, 88) = 6411
6	(74, 83, 76) = 6411	(64, 69, 91) = 6411	(68, 50, 94) = 6411
7	(64, 69, 91) = 6411	(69, 88, 85) = 6411	(68, 50, 94) = 6411
8	(69, 88, 85) = 6411	(64, 14, 96) = 6411	(63, 72, 91) = 6411
9	(5, 39, 72) = 3618	(63, 72, 91) = 6411	(63, 72, 91) = 6411
10	(2, 67, 73) = 3618	(68, 50, 94) = 6411	(80, 70, 88) = 6411

Таблица 4. Результаты работы алгоритм с метрикой ABC.

Метрика ABC			
Вариант\Поколение	0	1	99
1	(95, 27, 97) = 6351	(77, 36, 98) = 6351	(69, 46, 78) = 6351
2	(77, 36, 98) = 6351	(69, 46, 78) = 6351	(77, 36, 98) = 6351
3	(69, 46, 78) = 6351	(61, 65, 95) = 6351	(69, 46, 78) = 6351
4	(61, 65, 95) = 6351	(95, 27, 97) = 6351	(69, 46, 78) = 6351
5	(5, 67, 92) = 3538	(61, 65, 95) = 6351	(69, 46, 78) = 6351
6	(5, 87, 95) = 3538	(95, 27, 97) = 6351	(69, 46, 78) = 6351
7	(1, 35, 60) = 3538	(61, 65, 95) = 6351	(69, 46, 78) = 6351
8	(1, 70, 53) = 3538	(69, 46, 78) = 6351	(77, 36, 98) = 6351
9	(60, 30, 12) = 768	(69, 46, 78) = 6351	(69, 46, 78) = 6351
10	(60, 49, 73) = 768	(69, 46, 78) = 6351	(69, 46, 78) = 6351

5.4. Результаты с использованием метрики Джилба

В отличие от предыдущих метрик в данной учитывается абсолютная сложность программы, которая рассчитывается при делении количества циклов и условий на общее количество операций в пути. Сложность программы определяется совершенно иным способом, что привело к тому, что входные данные были подобраны для другого пути. Результаты представлены в таблице 5.

Таблица 5. Результаты работы алгоритм с метрикой ABC.

Метрика Джилба			
Вариант\Поколение	0	1	99
1	(75, 51, 3) = 100	(62, 25, 41) = 100	(78, 45, 21) = 100
2	(92, 33, 11) = 100	(94, 22, 35) = 100	(63, 36, 10) = 100
3	(94, 22, 35) = 100	(98, 51, 12) = 100	(75, 51, 3) = 100
4	(98, 51, 12) = 100	(80, 42, 20) = 100	(80, 42, 20) = 100
5	(80, 42, 20) = 100	(78, 45, 21) = 100	(78, 45, 21) = 100
6	(78, 45, 21) = 100	(80, 59, 8) = 100	(63, 36, 10) = 100
7	(80, 59, 8) = 100	(5, 40, 27) = 100	(80, 42, 20) = 100
8	(5, 40, 27) = 100	(99, 38, 29) = 100	(78, 45, 21) = 100
9	(99, 38, 29) = 100	(62, 25, 41) = 100	(75, 51, 3) = 100
10	(62, 25, 41) = 100	(63, 36, 10) = 100	(80, 42, 20) = 100

Полученные данные очень сильно различаются как с другими метриками, так и в пределах метрики. Связано это с особенностями тестируемого кода – в нём есть один общий цикл, внутри которого одно общее условие. Если данное условие не будет выполнено, то ни одна из операций, кроме цикла и условия, не будут учитываться при расчёте метрики. Значение 100 показывает, что среди всех операций на пути все являются циклами или условиями, т.е. формально подобранные входные данные являются вариантами, когда первое условие не было выполнено и другие операции не учитывались.

6. Заключение

Эволюционные методы работают таким образом, чтобы находить наиболее хорошие решения в задачах, которые невозможно или слишком затратно решать стандартными методами оптимизации. Они не всегда работают быстро или качественно, но в задачах с нестандартными подходами показывают превосходство.

Введение различных метрик для расчёта функции приспособленности позволил добавить алгоритму генерации входных тестовых данных большей вариативности и возможности вводить новые требования к данным. Каждая метрика сконцентрирована на определённых параметрах кода и может использоваться в случае, когда данные должны подбираться в соответствии с определёнными требованиями. Кроме того, в случае, когда метрика неэффективно подбирает данные, возможно использовать другие метрики, которые могут перекрывать недостатки друг друга.

Все анализируемые метрики, за исключением метрики Джилба, подобрали несколько вариантов данных для критического пути, который был выбран изначально. Заметно, что метрики для небольшого кода в 130 строк с несколькими путями кода успешно подбирают данные уже в первом поколении, что говорит о достаточно высокой скорости сходимости алгоритма. В последующих поколениях последовательно исключаются различные варианты.

Проведенные исследования позволяют предложить новый метод генерирования тестовых данных на основе генетического алгоритма, в котором функция приспособленности будет формироваться не на основе одной из известных метрик оценки сложности кода (как в данной работе), а на основе гибридной метрики, являющейся взвешенной суммой показателей, присутствующих в метриках, рассмотренных в данной работе. Также представляется перспективным в плане увеличения степени покрытия кода создание эффективного механизма регулирования (увеличения и снижения) весов операций в функции приспособленности при увеличении уровня вложенности участка кода.

В дальнейшем планируется расширить способы определения сложности кода. Кроме использования метрик непосредственно, планируется разработать метод для учёта показателей количества операций, функций, условий и циклов с различными весами. Так же возможно установить степени снижения или увеличения весов операций при разных уровнях вложенности. Это позволит задать приоритет для генерации входных при возникновении определённых требований.

7. Благодарности

Исследование выполнено при финансовой поддержке РФФИ в рамках научного проекта № 19-37-90156 «Разработка и исследование методов интеллектуального анализа и тестирования программного кода».

8. Литература

[1] Richard, A.D. Constraint-Based Automatic Test Data Generation / A.D. Richard, A.O. Jefferson // IEEE Transactions on Software Engineering. – 1991. – Vol. 17(9). – P. 900-910.

[2] Maragathavalli, P. Automatic Test-Data Generation for Modified Condition. Decision Coverage Using Genetic Algorithm / P. Maragathavalli, M. Anusha, P. Geethamalini, S. Priyadharsini // International Journal of Engineering Science and Technology. – 2011. – Vol. 3(2). – P. 1311-1318.

[3] Meudec, C. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution // Software Testing Verification and Reliability, 2001.
[4] Gerlich, R. Automatic Test Data Generation and Model Checking with CHR // 11th Workshop on Constraint Handling Rules, 2014.
[5] Girgis, M.R. Automatic Test Data Generation for Data Flow Testing Using a Genetic Algorithm // Journal of Universal Computer Science. – 2005. – Vol. 11(6). – P. 898-915.
[6] Weyuker, E.J. The complexity of data flow criteria for test data selection // Inf. Process. Lett. – 1984. – Vol. 19(2). – P. 103-109.
[7] Khamis, A. Automatic test data generation using data flow information / A. Khamis, R. Bahgat, R. Abdelaziz // Dogus University Journal. – 2011. – Vol. 2. – P. 140-153.
[8] Singla, S. A hybrid pso approach to automate test data generation for data flow coverage with dominance concepts / S. Singla, D. Kumar, H. M. Rai, P. Singla // Journal of Advanced Science and Technology. – 2011. – Vol. 37. – P. 15-26.
[9] Liu, Z. Hybrid Test Data Generation. State Key Laboratory for Novel Software Technology / Z. Liu, Z. Chen, C. Fang, Q. Shi // ICSE Companion 2014 Companion Proceedings of the 36th International Conference on Software Engineering, 2014 – P. 630-631.
[10] Harman, M. A Theoretical and Empirical Study of Search-Based Testing: Local, Global, and Hybrid Search / M. Harman, P. McMinn // IEEE Transactions on Software Engineering. – 2010. – Vol. 36(2). – P. 226-247.
[11] Xing, Y. A Hybrid Intelligent Search Algorithm for Automatic Test Data Generation / Y. Xing, Y. Gong, Y. Wang, X. Zhang // Mathematical Problems in Engineering, 2015.
[12] Paduraru, C. An Automatic Test Data Generation Tool using Machine Learning / C. Paduraru, M.C. Melemciuc // 13th International Conference on Software Technologies (ICSOFT), 2018. – P. 472-481.
[13] Boussaa, M. A Novelty Search Approach for Automatic Test Data Generation / M. Boussaa, O. Barais, G. Sunyé, B. Baudry // 8th International Workshop on Search-Based Software Testing, 2015.
[14] Lopez, M.A. DSL for Web Services Automatic Test Data Generation / M. Lopez, H. Ferreiro, L.M. Castro // 25th International Symposium on Implementation and Application of Functional Languages, 2013.
[15] Doungsa-Ard, C. An automatic test data generation from UML state diagram using genetic algorithm / C. Doungsa-Ard, K. Dahal, A.G. Hossain, T. Suwannasart // IEEE Computer Society Press, 2007. – P. 47-52.
[16] Sabharwal, S. Applying Genetic Algorithm for Prioritization of Test Case Scenarios Derived from UML Diagrams / S. Sabharwal, R. Sibal, C. Sharma // IJCSI International Journal of Computer Science Issues. – 2011 – Vol. 8(3(2)).
[17] Doungsa-Ard, C. GA-based Automatic Test Data Generation for UML State Diagrams with Parallel Paths / C. Doungsa-Ard, K. Dahal, A. Hossain, T. Suwannasart // Part of book Advanced design and manufacture to gain a competitive edge: New manufacturing techniques and their role in improving enterprise performance, 2008. – P. 147-156.
[18] Grochtmann, M. Classification trees for partition testing. Software Testing / M. Grochtmann, K. Grimm // Verification and Reliability. – 1993. – Vol. 3(2). – P. 63-82.
[19] Chen, T.Y. An integrated classification-tree methodology for test case generation / T.Y. Chen, P.L. Poon, T.H. Tse // International Journal of Software Engineering and Knowledge Engineering. – 2000. – Vol. 10(6). – P. 647-679.
[20] Cain, A. An Automatic Test Data Generation System Based on the Integrated Classification-Tree Methodology / A. Cain, T.Y. Chen, D. Grant, P.L. Poon, S.F. Tang, T.H. Tse // Software Engineering Research and Applications, Lecture Notes in Computer Science. – 2004 – Vol. 3026.
[21] Serdyukov, K. Investigation of the genetic algorithm possibilities for retrieving relevant cases from big data in the decision support systems / K. Serdyukov, T. Avdeenko // CEUR Workshop Proceedings. – 2017. – Vol. 1903. – P. 36-41.

[22] Praveen, R.S. Application of Genetic Algorithm in Software Testing / R.S. Praveen, K. Tai-Hoon // International Journal of Software Engineering and Its Applications. – 2009. – Vol. 3(4). – P. 87-96.
[23] Spears, W.M. Crossover or mutation? // Foundations of Genetic Algorithms 2, 1993. – P. 221-237.
[24] Mühlenbein, H. How genetic algorithms really work: Mutation and hillclimbing // Parallel Problem Solving from Nature 2, 1992.

Researching of methods for assessing the complexity of program code when generating input test data

K.E. Serdyukov[1], T.V. Avdeenko[1]

[1]Novosibirsk State Technical University, K. Marks avenue 20, Novosibirsk, Russia, 630073

Abstract. This article proposes a comparison of methods for determining code complexity when generating data sets for software testing. The article offers the results of a study for evaluating one path of program code, the work is not finished yet, it will be further expanded to select data for testing many paths. To solve the problem of generating test data sets, it is proposed to use a genetic algorithm with various metrics for determining the complexity of program code. A new metrics is proposed for determining code complexity based on changing weights of nested operations. The article presents the results and comparison of the generated input test data for the passage along the critical path. For each metric considered in the article, conclusions are presented to identify specifics depending on the selected data.