# Исследование возможностей применения генетического алгоритма для формирования наборов данных и первичной отладки программного кода

**К.Е. Сердюков[1], Т.В. Авдеенко[1]**

[1]Новосибирский государственный технический университет, Карла Маркса 20, Новосибирск, Россия, 630073

**Аннотация.** При разработке программного обеспечения даже самые высококвалифицированные разработчики допускают ошибки. Тестировщикам необходимо обеспечить достаточно высокий уровень покрытия кода, то есть проверку максимально возможного числа путей программы. Автоматизация тестирования возможна не на всех этапах. Один из этапов, который достаточно сложно поддаётся автоматизации, является подбор входных данных. Это связано с большим числом возможных комбинаций, поэтому в данной задаче особую ценность представляют эвристические подходы, одним из которых является генетический алгоритм. Внедрение генетического алгоритма позволяет найти достаточно большое число комбинаций, каждая из которых будет проверять определённыйпуть. За счёт настроек алгоритма можно обеспечить нахождение максимально удалённых друг от друга частей программного кода и получить более высокий уровень покрытия кода. В данной статье исследуются возможности разработанного метода генерации тестовых данных на основе генетического алгоритма.

## 1. Введение

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.

Некоторые процессы тестирования программного кода совершенствуются достаточно медленно. Разработка большинства видов дизайнов и шаблонов для 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. Именно эта задача решается в данной статье. В дальнейших следованиях решение задачи нахождения наиболее сложного пути предполагается экстраполировать для нахождения нескольких важных путей.

Исходя из вышесказанного можно сделать вывод, что автоматизация тестирования, или хотя бы автоматизация разработки тестов, может существенно снизить не только временные затраты, но и денежные. Есть и другие преимущества, не настолько очевидные – большая вероятность нахождения мелких ошибок, прозрачность разработки тестов, тестирование одновременно с разработкой программы и т.д. [1]

Масштабирование разработки привело к тому, что начали разрабатываться огромные программные системы, в которых участвуют множество разработчиков, каждый из которых обладает собственным стилем программирования и разными компетенциями. Несмотря на то, что вместе с этим появились системы, позволяющие обеспечить высокий уровень совместной разработки, с контролем за изменениями и возможностями проверки качества кода, финальный продукт далеко не всегда соответствует заданным на этапе планирования требованиям.

По этой причине существенно возрастает необходимость качественного и всестороннего тестирования. Необходимо не только находить ошибки в программном коде, но и логические нестыковки. Чтобы более качественно протестировать как программу в целом, так и отдельные её части, необходима не только команда тестировщиков, но и существенная подготовительная деятельность – определение набора входных данных, которые протестировали бы определённые части программы [2].

Тестирование не является стандартизированным процессом, оно зависит от многих факторов, большая часть которых меняется от одной программы к другой. Поэтому учёные пришли к выводу об использовании эвристических подходов, в частности, на основе искусственного интеллекта, которые бы позволили создать гибридную систему автоматической разработки тестов.

Определённое место в решении данной задачи занимают генетические алгоритмы. В работе [3] проводится сравнение различных методов генерации тестовых данных, в том числе генетических алгоритмов, метода случайного поиска и других эвристических методов. В статье [4] для решения данной проблемы предлагается использовать метод на основе логического программирования в ограничениях (анг. Constraint Logic Programming) и символьных вычислений (Symbolic Execution). В [5] 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 с использованием диаграммы потоков данных (анг. Data-flow diagram). Исследования методов автоматизирования с использованием данной диаграммы были предложены в статьях [6-7]. В статье [8] предлагается дополнительно использовать генетические алгоритмы с для определения новых входных тестовых наборов данных на основе ранее использованных.

В работах [10-12] рассматриваются гибридные автоматизированные системы генерации тестовых данных. Так, в [9] применяется подход, объединяющий стратегии случайного поиска (анг. Random Strategy, RS), динамических символьных вычислений (анг. Dynamic Symbolic Execution, DSE) ипоисковаястратегий (анг. Search-Based Strategy, SBS). В статье [13] предлагается теоретическое описание поисковой стратегии тестирования с применением генетического алгоритма. Рассматриваются подходы поиска локальных и глобальных экстремумов на реальных программах. Предлагается гибридный подход генерации тестовых данных – имитационный алгоритм (анг. MemeticAlgorithm).

В статье [14] используется гибридный интеллектуальный алгоритм поиска для генерации тестовых данных. В предлагаемом алгоритме используется методы ветвей и границ (анг. BranchandBound) и поиска экстремумов (анг. Hillclimbing) с использованием интеллектуального поиска.

Существует множество исследований по теме генерации тестовых данных. Чаще всего для решения данной задачи применяются эвристические подходы, так как они позволяют подобрать данные не полным перебором всех возможных вариантов. Предложенный в данной статье подход основан на генетическом алгоритме с модификацией расчёта функции приспособленности, который позволяет сгенерировать данные на основе программного кода без привязки к каким-либо системам тестирования и разработки. Это позволяет генерировать данные напрямую, лишь указав ограничения на входные переменные, не ограничиваясь особенностями кода, без необходимости заранее определять тестовые наборы. Алгоритм может быть улучшен для применения не только в задаче генерации данных, но и для помощи в отладке программы.

## 2. Генетические алгоритмы

Генетический алгоритм является эвристическим методом, более точно – одной из разновидностей эволюционных алгоритмов, который использует идею и терминологию из естественной эволюции. Его цель заключается не в нахождении оптимального (самого лучшего) решения, а в нахождении решения, достаточно близкого к оптимальному. Поэтому данный алгоритм не рекомендуют применять, если уже существуют быстрые и хорошо проработанные методы оптимизации. При этом, генетический алгоритм отлично показывают себя в решении нестандартных задач, задач с неполными данными или задач, для которых невозможно применение оптимизационных методов из-за сложности реализации или длительности выполнения [15, 16].

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. Понятие функции приспособленности, является одним из самых важных во всём методе. Как правило, генетический алгоритм решает задачи максимизации или минимизации и адекватность каждого решения (хромосомы) оценивается с помощью функции приспособленности.

Генетический алгоритм работает последующему принципу:

- *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 определённое значение, оценивающее, насколько хорошо они соответствует решаемой проблеме [17].
- *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.* Первое существенно отличие от обычных методов и один из важнейших шагов во всем алгоритме. 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 замены частей друг друга. После того, как заполнится необходимое число хромосом для создания популяции, алгоритм переходит к следующему шагу [18].
- *Мутация.* Это также шаг, характерный только для ГА. В случайном порядке случайный ген может поменять значения на случайное. Основное предназначение мутации такое 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 на совершенно другойпутьи, во-вторых, «разбавить» популяцию, чтобы избежать ситуации, когда во всей популяции будут только одинаковые хромосомы, которые не будут двигаться к нахождению глобального решения.

- После того, как проведены все этапы генетического алгоритма, оценивается, достигла ли популяция желаемой точности решений, или было достигнуто ограничение 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 отдельной операции [19].

На сегодняшний день разработано множество видов диаграмм, которые позволяют представить структуру программы не в виде набора действий, а в виде диаграмм с определённой структурой. Одним из их видов являются диаграммы (графы) потоков управления, позволяющие представить всё множество путей выполнения программы. Основное назначение подобных диаграмм приходится на задачу проектирования программы – для определения сложности описываемого кода, проверки логики и непосредственного определения шагов выполнения программы. Однако, если посмотреть с позиции проблемы формирования тестовых данных, то подобный тип диаграмм, построенный уже на реально созданной программе, позволяет оценить качество разработанного кода и, в рамках задачи, оценить важность, или сложность, определённых путей программы.

Основываясь на возможности представления программы в структурном виде, был разработан метод, по которому можно будет оценить программный код и определить такой набор тестовых данных, который бы позволил «пройтись» по наибольшему числу операций и наибольшему числу путей. Первым шагом является рассмотрение структурных элементов кода. Для удобства представления можно использовать диаграммы потоков, чтобы визуализировать структуру кода и понять, каким образом происходит выполнение программ.

Каждому действию присваивается свой отдельный узел, а в качестве связей выступает направление при выполнении кода. Например, условию присуждается один узел, но из него будут выходить 2 пути кода. Каждому переходу между узлами назначается определённый вес в зависимости от того, в какой именно части кода находится данное действие, предшествуют ли ему какие-нибудь сложные структурные элементы и т.д. [20].

В задаче по поиску входных тестовых данных можно выделить следующие подзадачи:

1) Поиск входных данных для прохода по одному наиболее сложному пути кода;
2) Исключение или снижение весов операций этого пути из расчёта последующих путей;
3) Поиск набора тестовых данных для прохода по множеству путей.

Ограничение на размер набора входных данных устанавливается после этапа разработки и позволит сконцентрироваться на определённых путях, в которых выполняется наибольшее количество операций.

Весь алгоритм работает циклически – запускается процедура поиска входных данных для одного пути, после чего операции этого пути исключаются из дальнейших вычислений, и алгоритм снова запускается для поиска данных одного пути.

Поиск по одному пути работает следующим образом:

- Первой операции назначается вес, например, в 100 единиц.
- Каждой последующей операции также назначается вес – если нет никаких условий или циклов, вес приравнивается весу предыдущей операции.
- 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 ветвей 50%/50%, для трёх 33%/33%/33% и т.д.

- Веса операций в цикле остаются, но также могут умножаться на определённый вес, если необходимо увеличить значимость циклов при тестировании.
- Все вложенные ограничения учитываются, например, для двух вложенных условий вес операций будет равен 80% * 80% = 64%

Пример, на основе которого проверялся алгоритм, представлен на рисунке 1.

Назначенные веса можно использовать для того, чтобы разрабатывать тестовые варианты при помощи генетических алгоритмов, то есть для оценки того, какой вес приходится на тот или инойпуть 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) →*max*.

```
0.  public static void Main(int m, int i, int n) {
1.  intsum = 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. }
```

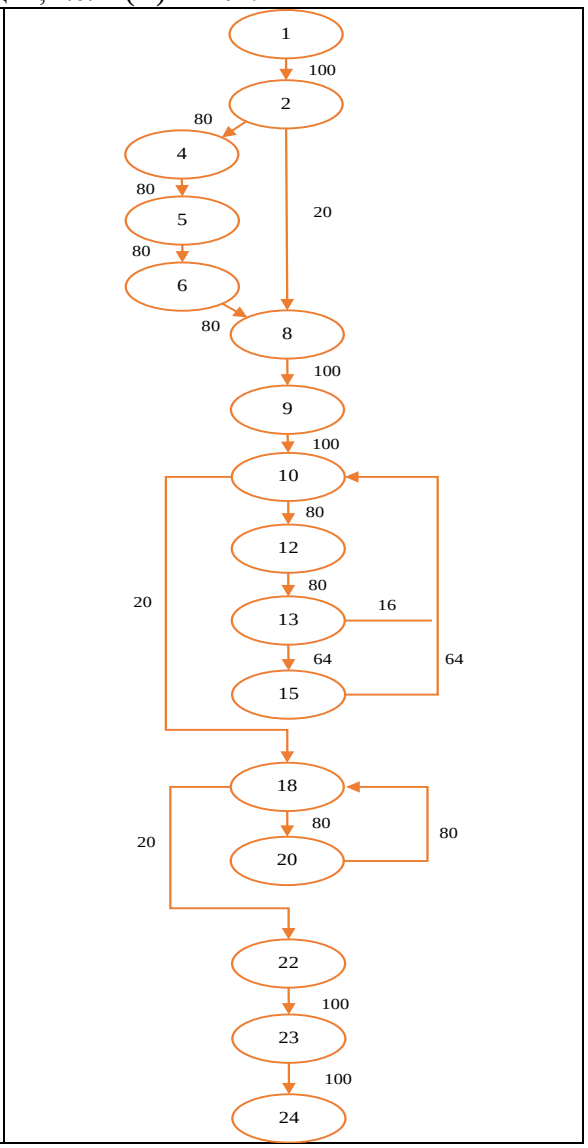


**Рисунок 1**. Программный код и последовательность графов анализируемой программы.

## 4. Представление результатов

Для более подробного рассмотрения алгоритма используем простые тестовые варианты и проведём каждый из шагов вручную с подробным описанием действий. На этапе инициализации сформируем следующие наборы данных – (10,5,12); (3,4,10); (25,30,11); (5,3,17);

В таблице 1 представлены эти наборы, рассчитанное целевое значение функции приспособленности и ранг, определяющий наилучший набор.

**Таблица 1**. Первоначальные варианты наборов данных.

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

В данном случае в качестве наборов данных для селекции будут использоваться 2 и 3 варианты. Для того, чтобы получить дополнительно два новых варианта, их значения будут перемешаны и дополнены некоторой вероятностью мутации.

Деление наборов будет происходить по первой и второй позиции – при скрещивании (R1,R2,R3) и (G1,G2,G3) будут получены переменные – (R1,R2,G3), (R1,G2,G3), (G1,R2,R3) и (G1,G2,R3). Родительские значения остаются для сохранения чистоты скрещивания, т.е. по сравнению с нулевым поколением в первом будут добавлены 2 новых набора. В последующих поколениях будет использоваться 6 наборов данных. Стоит оговориться, что в зависимости от настроек, родительские хромосомы могут исключаться из рассмотрения.

Мутация будет происходить с вероятностью в 10% на шанс изменения значения от 1 до заданного значения в обе стороны. В данных условиях максимально возможное значение добавления значения при мутации составляет 5. В результате скрещивания, будут получены наборы данных, отражённые в таблице 2.

**Таблица 2.** Новые наборы данных, полученные в результате скрещивания.

| № | X | Y | Новый набор | **Мутация** |
|---|---|---|---|---|
| 1 | (3,4,10) | (25,30,11) | (3,4,11) | **(3,4,13)** |
| 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)** |

В результате к дополнительным двум родительским наборам добавятся ещё два - (3,4,13) и (25,30,10). В таблице 3 показаны все новые варианты тестовых наборов данных.

**Таблица 3.** Первое поколение тестовых данных.

| № | Набор данных X | F(X) | Ранг | Поколение |
|---|---|---|---|---|
| 1 | (3,4,10) | 1196 | 2 | 0 |
| 2 | (25,30,11) | 1308 | 1 | 0 |
| 3 | (3,4,13) | 1196 | 2 | 1 |
| 4 | (3,30,11) | 1308 | 1 | 1 |
| 5 | (25,4,10) | 896 | 3 | 1 |
| 6 | (25,30,10) | 1308 | 1 | 1 |

При одинаковых рангах приоритет будет у наборов из более нового поколения. В последнем поколении было получено 3 набора данных, которые проверяют больше всего путей программы - (25,30,11), (3,30,11) и (25,30,10). Первый набор был получен из первого поколения, поэтому он будет исключён и останется два варианта – (3,30,11) и (25,30,10).

Из-за маленькой начальной выборки и небольшого кода наборы данных достаточно быстро пришли к нахождению пересекающихся значений – 30 на второй позиции и 10 или 11 на третьей. Поэтому продолжать проводить итерации перестаёт иметь смысл – уже в следующем поколении данные будут состоять преимущественно из повторяющихся наборов.

Для текущего программного кода можно использовать тестовые наборы данных, полученных в последнем поколении. Приоритетность зависит от полученного ранга.

Таким образом, используя генетические алгоритмы можно найти такие начальные тестовые начальные значения, которые бы в полной мере проверяли все пути программы.

## 5. Усовершенствование алгоритма

Алгоритм позволяет не только оценить, какие пути программы будут задействованы при определённых данных, но и каким образом данные меняются, и сохраняются ли повторяющиеся значения у лучших хромосом между популяциями.

Это не только даёт возможность определить начальный тестовый набор, но и на основе анализа данных сделать выводы о наличии логики в программе, которая соответствует запланированной.

Для представления результатов было проведено четыре дополнительных теста. Первая популяция формируется случайными значениями. В каждой популяции содержится по 100 хромосом. Общее число популяция также равняется 100.

В таблице 4 представлены итоги исследования.

**Таблица 4**. Сравнение результатов запуска метода.

| Популяция | Тест 1 | Тест 2 | Тест 3 | Тест 4 |
|---|---|---|---|---|
| 0 | 1: 78, 23, 35<br>2: 62, 36, 95<br>3: 52, 35, 27<br>4: 17, 77, 73<br>5: 75, 9, 96 | 1: 97, 3, 6<br>2: 82, 77, 64<br>3: 24, 47, 57<br>4: 90, 13, 82<br>5: 81, 69, 24 | 1: 92, 97, 28<br>2: 38, 66, 52<br>3: 63, 76, 64<br>4: 7, 24, 56<br>5: 57, 48, 8 | 1: 15, 67, 26<br>2: 32, 27, 83<br>3: 37, 52, 64<br>4: 70, 49, 64<br>5: 67, 29, 94 |
| 20 | 1: 95, 64, 54<br>2: 95, 64, 29<br>3: 95, 64, 54 | 1: 97, 80, 4<br>2: 97, 80, 53<br>3: 97, 80, 28 | 1: 99, 13, 10<br>2: 99, 13, 11<br>3: 99, 13, 11 | 1: 99, 71, 45<br>2: 99, 71, 15<br>3: 99, 71, 3 |
| 50 | 1: 95, 64, 54<br>2: 95, 64, 29<br>3: 95, 64, 54 | 1: 97, 80, 29<br>2: 97, 80, 4<br>3: 97, 80, 53 | 1: 99, 13, 10<br>2: 99, 13, 11<br>3: 99, 13, 11 | 1: 99, 71, 60<br>2: 99, 71, 3<br>3: 99, 71, 3 |
| **Итоговая (100)** | **1: 95, 64, 54<br>2: 95, 64, 29** | **1: 97, 80, 4<br>2: 97, 80, 29** | **1: 99, 13, 10<br>2: 99, 13, 11** | **1: 99, 71, 60<br>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. Так, первое значение всегда максимально (случайные значения ограничивались значением 100). Второе значение меньше, чем первое, но больше 3-го.

Для дополнительного анализа проверим, как именно будет меняться скорость работы алгоритма в зависимости от настроек ГА. На графиках ниже можно увидеть, каким образом меняется длительность работы программы в зависимости от размера популяции, т.е. количества хромосом, и от общего количества популяций. При исследовании изменения размера популяций количество популяций приравнивалось к 100. И наоборот, при исследовании зависимости от количества популяций, число хромосом равнялось 100.

На рисунке 2 показывается зависимость скорости работы метода от количества хромосом в популяции. Основываясь на нём, можно сделать вывод, что при увеличении количества хромосом в популяции длительность работы алгоритма существенно увеличивается, причём по экспоненте.

На рисунке 3 показываться зависимость от количества популяций.

На рисунках видно, что при изменении количества популяций увеличивается длительность работы алгоритма, но уже в линейной зависимости. При этом заметны колебания прироста в обе стороны, сохраняющиеся примерно на одном уровне.

Несмотря на то, что и в одном, и в другом случае общее количество хромосом оставалось одинаковым, изменение в скорости работы алгоритма существенно различается. Увеличение размера популяции существенно замедляет работу алгоритма – число хромосом в популяции с

1 000 до 50 000 увеличилось в 50 раз, но длительность возросла в 750. Изменение же количества популяций в 50 раз привело к увеличению времени лишь в 14 раз.

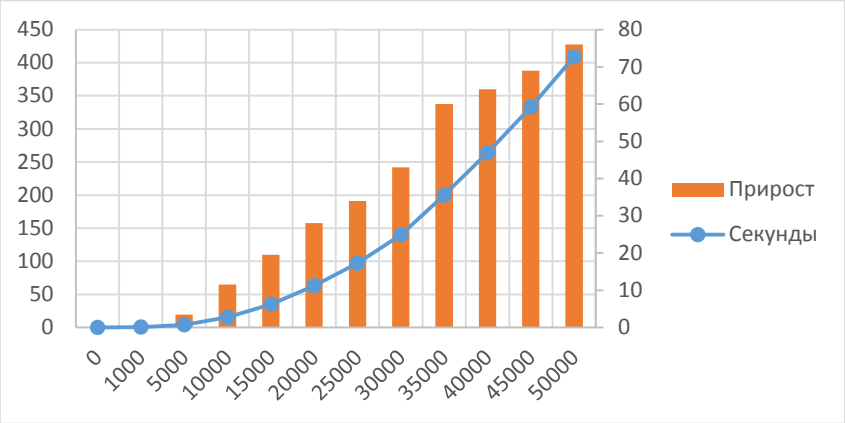


**Рисунок 2.**Зависимость длительности выполнения от размера популяций.

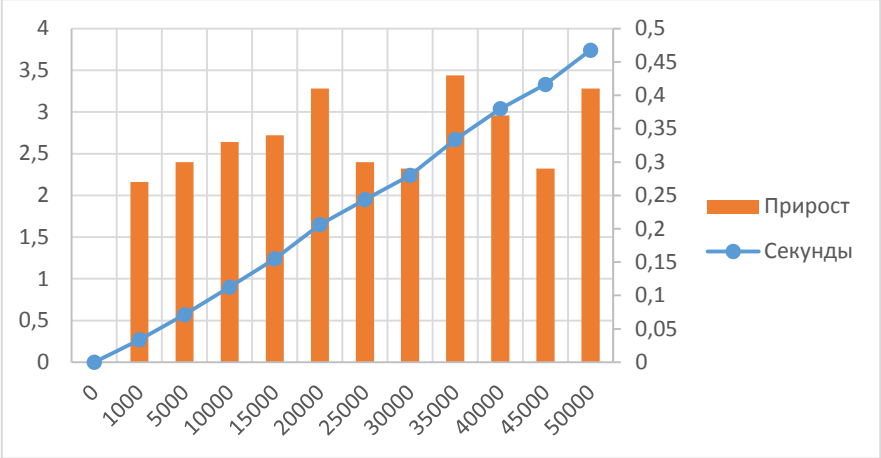


**Рисунок 3.**Зависимость длительности выполнения от количества популяций.

Связано это с тем, что наиболее сложные с точки зрения нагрузки операции происходят при расчёте функции приспособленности и при поиске оптимальных для скрещивания хромосом. Именно из-за того, что поиск лучших хромосом зависит от числа хромосом в популяции, то текущий алгоритм поиска, т.е. сортировки от лучших к худшим, существенно загружает мощности компьютерных систем и увеличивает скорость работы экспоненциально размеру популяции.

Количество хромосом в одной популяции позволяет обеспечить разнообразие вариантов, т.е. с большей вероятностью найти более подходящие варианты. Увеличение количества популяций приводит к более точному результату, но только при большом количестве хромосом. Если хромосом будет достаточно мало, то алгоритм быстро придёт к одному повторяющемуся значению.

Исходя из-за всего вышесказанного, наибольшее влияние на результат оказывает число хромосом в одной популяции, но при этом при увеличении этого количества существенно возрастает нагрузка. Но чтобы обеспечить более качественный конечный результат, при увеличении числа хромосом также следует увеличивать общее число популяций, что также влияет на загрузку.

## 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.

Метод подбора входных данных на основе генетического алгоритма позволит автоматизировать метод подбора входных данных, при этом существенно увеличив скорость поиска данных. За исключением настроек ограничений, алгоритм полностью автоматизирован, поэтому не требует дополнительных трудовых затрат. Полученные на выходе наборы данных можно напрямую использовать в процессе тестирования. При необходимости новые наборы можно получить заново без дополнительных затрат.

В дальнейшем, алгоритм может быть объединён с визуализаторами на основе диаграмм, чтобы понимать, на какой именно путь кода будет выводить тот или иной набор данных и, соответственно, понимать, в какой именно части кода результаты не соответствуют плановым значениям, что, несомненно, ускорит не только сам процесс тестирования, но и последующее устранение ошибок разработчиками.

В дальнейшем планируется провести исследование влияния различных методов расчёта сложности программы, которые используются в функции приспособленности, на конечный результат и величину покрытия кода (число протестированных операций, делённое на общее число операций), которые бы обеспечили такие наборы данных, позволяющие протестировать код максимально эффективно и с максимальным числом учитываемых операций.

## 7. Литература


[1] Zanetti, M.C. Automated Software Remodularization Based on Move Refactoring. A Complex Systems Approach / M.C. Zanetti, C.J. Tessone, I. Scholtes, F. Schweitzer // 3th international conference on Modularity, 2014. – P. 73-83

[2] Crispin, L. Agile Testing: A Practical Guide for Testersand / L. Crispin, J. Gregory // Agile Teams. – Pearson Education, 2010. – 576 p.

[3] 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.

[4] Meudec, C. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution // Software Testing Verification and Reliability, 2001.

[5] Gerlich, R. Automatic Test Data Generation and Model Checking with CHR // 11th Workshop on Constraint Handling Rules, 2014.

[6] Moheb, 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.

[7] Weyuker, E.J. The complexity of data flow criteria for test data selection // Inf. Process. Lett. – 1984. – Vol. 19(2). – P. 103-109.

[8] 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.

[9] 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.

[10] Luger, F. George Artificial Intelligence Structures and Strategies for Complex Problem Solving // University of New Mexico, 2009. – 679 p.

[11] Berndt, D.J. Breeding Software Test Cases with Genetic Algorithms / D.J. Berndt, J. Fisher, L. Johnson, J. Pinglikar, A. Watkins // Proceedings of the Thirty-Sixth Hawaii International Conference on System Sciences, 2003. – Vol. 36.

[12] Liu, Z. Hybrid Test Data Generation / Z. Liu, Z. Chen, C. Fang, Q. Shi / State Key Laboratory for Novel Software Technology // ICSE Companion Proceedings of the 36th International Conference on Software Engineering, 2014. – P. 630-631.

[13] 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.

[14] Xing, Y. A Hybrid Intelligent Search Algorithm for Automatic Test Data Generation / Y. Xing, Y.Z. Gong, Y.W. Wang, X.Z. Zhang // Mathematical Problems in Engineering. – 2015. – Vol. 2015. – P. 15.
[15] 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.
[16] Yang, H.L. Two stages of case-based reasoning - Integrating genetic algorithm with data mining mechanism / H.L. Yang, C.S. Wang // Expert Systems with Applications. – 2008. – Vol. 35. – P 262-272.
[17] Mühlenbein, H. How genetic algorithms really work: Mutation and hillclimbing // Parallel Problem Solving from Nature 2, North−Holland, 1992.
[18] Spears, W.M. Crossover or mutation? Foundations of Genetic Algorithms 2, 1993.
[19] 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.
[20] Coyle, L. Improving recommendation ranking by learning personal feature weights / L. Coyle, P. Cunningham // Proc. 7th European Conference on Case-Based Reasoning, 2004. – P. 560-572.

**Благодарности**
Работа поддержана грантом 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».

# Researching of using genetic algorithm for generating data sets and initial debugging of program code

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

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

**Abstract.** Even most qualified programmers in process of software developing make mistakes. Testers need to ensure a sufficiency level of code coverage with maximum possible numbers of program branches. Automatization testing is not possible at all stages. One of the stages which is quite difficult to automate is generation of input data. Due to large number of possible data combination heuristic approaches have particular value, one of which is the genetic algorithms. Advert of the genetic algorithm allow to find fairly large number of combinations, each of which handle a particular branch. By adjusting the algorithm, it is possible to ensure that parts of code which is most distant from each other are found and higher level of code coverage is obtained. This article explores the possibilities of the developed method for generating input test data based on genetic algorithm.