Метод применения генетического алгоритма для автоматической генерации тестовых данных

К.Е. Сердюков[1], Т.В. Авдеенко[1]

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

Аннотация. Тестирование программного обеспечения всегда было достаточно трудозатратным процессом, при этом не несущим в себе очевидной выгоды и результатов, но являющимся не менее важным, чем любая остальная часть программной инженерии. Одной из основных частей тестирования является разработка начальных тестовых данных, которые позволяют адекватно оценить программный код и найти наиболее важные части программы. Методы интеллектуальных систем, а в особенности генетический алгоритм, позволяют определять наборы данных, которые в полной мере позволят оценить значения переменных для качественного тестирования на их основе. В данной статье определяется метод возможной гибридизации генетического алгоритма и системы определения тестовых данных.

1. Введение

Одним из наиболее важных шагов при разработке программных продуктов является тестирование. Соответствие разработанной программы заданным требованиям, соблюдение логики в процессах обработки данных и получение верных конечных результатов – всё это входит в цели тестирования.

Процессы верификации и валидации 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 главных целей тестирования является создания такого тестового набора, который обеспечивал бы достаточный уровень качества конечного продукта, за счёт проверки большинства различных ветвей программного кода.

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]

Очевидно, что при указанных преимуществах процесса автоматизации тестирования, исследователи предлагают различные подходы к указанному процессу. Например, в работе [[2] предлагается автоматизация программирования сложных производственных систем в соответствии со стандартом IEC 61131.

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 некоторых исследований в области искусственного интеллекта, которые бы позволили создать гибридную систему автоматической разработки тестов[[3,4].

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 [6, 7].

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 была получена 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:

- 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 [8]. 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 [9]. 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. 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 различные веса каждой отдельной операции [5].

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. Но 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 ветвей программы.

В соответствии с этим 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 ветвей. Первым шагом является рассмотрение структурных элементов кода. Для удобства представления можно использовать диаграммы потоков, о которых было сказано ранее, чтобы визуализировать структуру кода и понять, каким образом происходит выполнение программ

Каждому действию присваивается свой отдельный граф, а в качестве связей выступает направление при выполнении кода. Например, условию присуждается один граф, но из него будет выходить 2 ветви кода. Каждому переходу между графами назначается определённый вес в зависимости от того, в какой части кода именно находится данное действие, предшествуют ли ему какие-нибудь сложные структурные элементы и т.д. [10]

Вес каждого перехода определяется при помощи правила Парето – 80% веса приходится на сложную ветвь кода, а 20% на простую. При одинаковом значении «сложности» разделение происходит 50%/50%. Для примера, если в коде будет обычное условие, с одной ветвью при положительном результате сравнения, на действия данного условия будет приходится 80% веса, а 20% на ветвь, ведущую к продолжению. Но если в условии присутствуют обе ветви, то есть при выполнении и невыполнении условия, то тогда веса будут распределены 50% на 50%.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, 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. Представление результатов

Для тестирования метода используется 4 случайно сформированных набора данных – (10,5,12); (3,4,10); (25,30,11); (5,3,17);

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

Таблица 1. Первоначальные варианты наборов данных.

№	Набор данных X	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

```
0.  public static void Main(int m, inti, 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. }
```

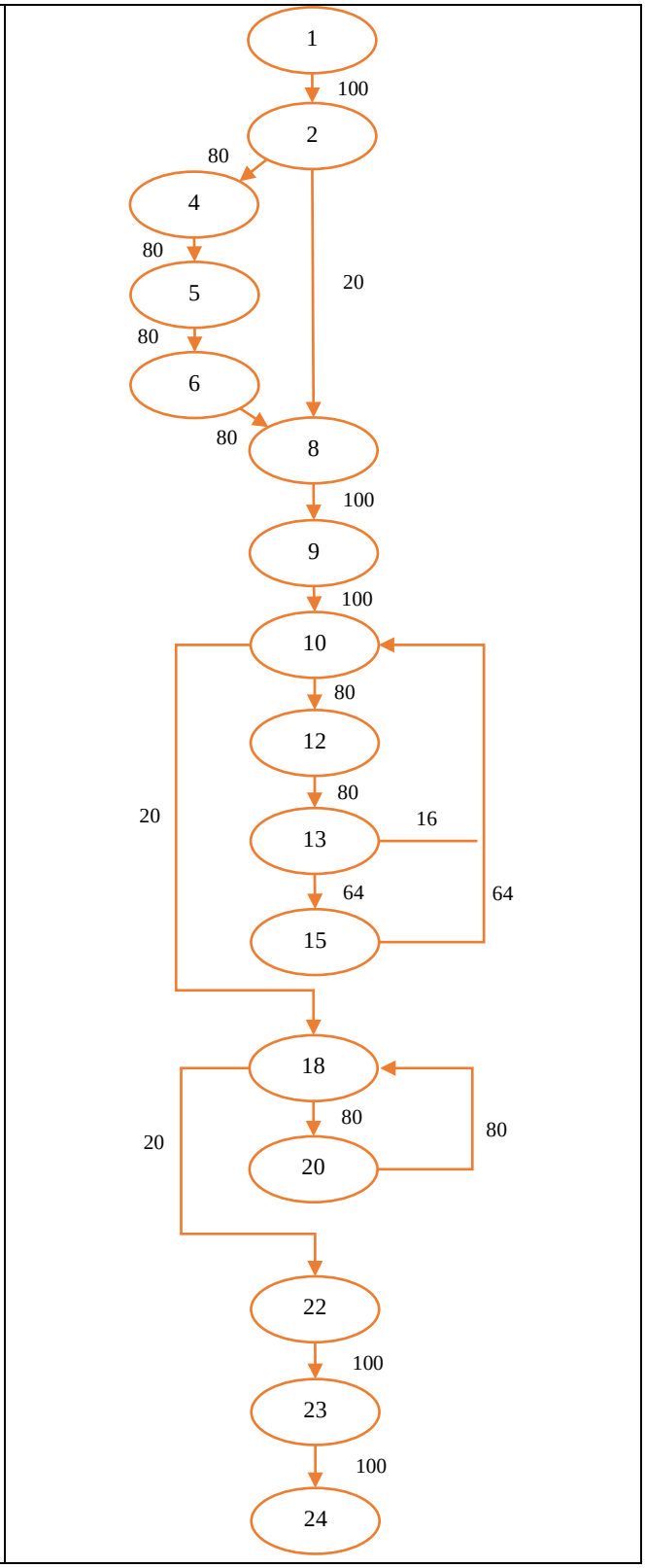

Рисунок 1. Программный код и последовательность графов анализируемой программы.

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 (X,X,X) и (Y,Y,Y) будут получены переменные – (X,X,Y), (X,Y,Y),(Y,X,X)и (Y,Y,X). 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 сохраняться 6 наборов данных.

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, будет 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 итоге 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 ветвей программы - (25,30,11), (3,30,11) и (25,30,10). Первый набор был получен из первого поколения, поэтому он будет исключён и в результате 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 на третьей. Поэтому продолжать проводить итерации перестаёт иметь смысл – уже в следующем поколении наборы будут состоять преимущественно из повторяющихся наборов.

Для данного 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 варианты программы в зависимости от назначенных весов. Это проверяется через максимизацию функций приспособленности, так как наиболее широко описывающий тестовый вариант проходит через те ветви, которые имеют наибольший вес.

В качестве значений хромосом выступают непосредственно сами значения тестовой выборки. Множество тестовых вариантов в одной итерации становятся популяцией.Используя генетический алгоритм становится возможным просмотреть множество тестовых выборок для нахождения наилучших начальных вариантов.

5. Усовершенствование алгоритма

В соответствии с ранее рассмотренным вариантом использования, 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. Он

позволяет не только оценить, какие ветви программы будут задействованы при таких данных, но и как они изменяются непосредственно при работе программы и какие будут получены результаты. Это не только даёт возможность определить начальный тестовый набор, но и на основе конечной популяции сделать выводы об изменении данных и их модификации в соответствии с заранее заданной логике. Для представления результатов было проведено 4 дополнительных тестов.

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 4 представлены итоги тестирования.

Таблица 4. 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 ветвям. 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.

6. Заключение

Задачу по определению наиболее подходящих тестовых данных очень сложно решить при помощи стандартных алгоритмов, так как фактическое количество всего массива возможных значений невероятно огромно, и подобрать действительно подходящие перебором значений не представляется возможным. При этом, автоматизация данного процесса может существенно уменьшить аналитическую нагрузку на пользователей, занимающихся тестированием.

Использование генетических алгоритмов позволяет сравнивать множество различных вариантов данных для тестирования программы. Широкие возможности к усовершенствованию позволяет увеличить количество начальных тестовых вариантов, количество поколений и добавить новые свойства, благодаря которым можно существенно увеличить возможности нахождения более подходящих вариантов. Если отслеживать пройденные графы и снижать веса тех графов, которые наиболее часто встречаются в различных вариантах можно обеспечить поиск новых путей, которые на данный момент могут не попадаться, но могут быть важны не менее, чем наиболее часто встречающиеся.

Сгенерированные тесты могут служить не только для тестирования алгоритма, но анализа его работоспособности. Данные в своём чистом виде уже позволяют определить закономерности программного кода, а при более глубоком изучении могут позволить сделать выводы о возможном последующем улучшении. Такой анализ может быть использован в качестве темы дальнейших исследований в области анализа данных.

Кроме этого, необходимо решить несколько неоднозначных задач, из-за которых простое назначение весов не сможет обеспечить качественный поиск ветвей программы:

- Переходы между различными частями программ. Например, в объектно-ориентированном программировании это может быть запуск других подклассов, каждый

из которых имеет собственные пути, запуск других функций и расчёт изменений в значениях. Множество рекомендаций для программистов относится именно к тому, чтобы наиболее трудные для понимания части выносились в отдельные процедуры, что существенно затрудняет нахождение действительно важных частей программы.

- Циклы, которые существенно усложняют адекватную оценку программного кода, ведь некоторые могут занимать огромную долю в ресурсных затратах, но при этом иметь достаточно простую логику. В итоге становится сложно оценить, является ли какой-либо определённый цикл важной частью программы и в действительности сколько итераций может быть запущено при определённых условиях.
- Недостаточная проработка кода. Часто ограничение на ввод вносится вне кода, из-за чего при использовании генетического алгоритма может потенциально попасться такой набор данных, при которых программа никогда не выполнится. С одной стороны, это поможет находить такие проблемные места, а с другой ставит ограничение непосредственно на алгоритм.

Это не единственные проблемные задачи, из-за которых появляются трудности к правильной оценке работоспособности программы. Тем не менее, их решение позволит не только выполнить текущую заданную цель предложенного метода, но и получить дополнительные преимущества от его использования, таких как, определение неоптимальных частей кода, лишних операций, «замусоренность» кода и т.п. Как итог, дальнейшие исследования позволят получить новые, заранее не предопределённые результаты.

7. 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».

8. 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] John, K. Programming Industrial Automation Systems: Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Aids / K. John, M. Tiegelkamp // Springer, Berlin, 2010. – P. 388.

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

[4] 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. – HICSS-36, 2003.

[5] 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.

[6] 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.

[7] 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.

[8] Mühlenbein, H. How genetic algorithms really work: Mutation and hillclimbing / H. Mühlenbein // Parallel Problem Solving from Nature 2. – North−Holland, 1992.

[9] Spears, W.M. Crossover or mutation? / W.M. Spears // Foundations of Genetic Algorithms 2, 1993.

[10] 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.

Method of application the genetic algorithm for automatic generation of test data

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

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

Abstract. Software testing has always been a time-consuming process, without obvious results but not less important than any other part of software engineering. One of the main part of testing is the development of initial test data that allow complete evaluate the program codeand find most important parts of the program. Methods of intelligent systems in particular genetic algorithm allow to define initial data set that allow evaluate the values of variables for quality testing. This article determines the method of possible hybridization of genetic algorithm and system of defining test data.

Keywords: intelligent systems, testing, software engineering, genetic algorithm.