DYNAMIC-RADIUS SPECIES-CONSERVING GENETIC ALGORITHM FOR TEST GENERATION FOR STRUCTURAL TESTING

Michael Scott Brown and Michael J. Pelosi

Information Technology and Serveries Department, University of Maryland University College, Adelphi, Maryland, United States

ABSTRACT

Software testing is a critical and labor-intensive activity in software engineering. Much research has been done to help automate test case generation. This research proposes a new approach to structural test case generation. It uses a specialized genetic algorithm called Dynamic-radius Species-conserving Genetic Algorithm (DSGA) to find a structurally complete set of test cases for the Triangle Classification algorithm. DSGA is a Niche Genetic Algorithm (NGA) that uses a short-term memory structure to store optima. Each individual of the NGA represents the inputs for a test case. The fitness function encourages the algorithm to locate test cases that cover large areas of the structure of the program. A shared fitness encourages the NGA to locate other areas of the structure. DSGA is a novel approach to structurally complete test case generation.

KEYWORDS

Niche Genetic Algorithm, Genetic Algorithm, Software Testing, Automated Test Case Generation, Structural Testing

1. INTRODUCTION

Software testing can be very time consuming and difficult. Some studies show that as much as 50% of software development effort is used to test software [1]. Much of this time is spent generating test cases that can meet a testing objective. Much research has been conducted to automate software testing. Research covers generated test data for all types of testing like structural test cases [2], unit testing [3] and functional testing [4]. Automation of testing allows for software to be developed quicker and at a lower cost through reduced cost in manual generation of test cases.

Structural Testing is one of many categories of software test cases. In Structural Testing the goal is to test all areas of the program based upon the structure of the program. The simplest form of Structural Testing is Statement Testing. In Statement Testing the goal is to obtain a set of test cases that execute every statement in the program. A set that executes every statement is said to be statement complete [5]. In most software not every statement can be executed. Exception handling is a primary example. In these cases it can be stated that some percentage of statement coverage has been achieved. In addition to statement coverage, Structural Testing can also be done for paths, branches, methods, classes and other structural characteristics.

One method often used for software testing automation is the Genetic Algorithm (GA) [6]. GAs are a category of optimization algorithms that work very well at searching large complex domain spaces. In software testing GAs have been used for test case reduction [7], input test generation [8] and even fixing software defects [9]. GAs demonstrably perform well in the automation of software testing.

This paper presents a new GA for the automation of input test generation for structural testing. This research uses a highly specialized GA called the Dynamic-radius Species-conserving Genetic Algorithm (DSGA) to generate input data for structural completeness [10]. DSGA for Structural Tests (DSGA-ST) works in a manner that is similar to how humans generate test cases for structural testing. It begins by locating some input values and determines what parts of the structure they are testing. Then it excludes those areas of the domain to search for new inputs that test other parts of the structure. This approach helps the GA quickly locate input data for test cases.

This research makes a number of contributions to the area of structural testing field research. DSGA-ST develops an entire test suite with a single pass running of a GA, while other methods have to run GAs multiple times for each condition within the target program. Algorithms of this type generate completely new fitness functions for each condition within the target program. Also, with DSGA-ST the size of the test suite does not need to be known. Some other algorithms model the test suite as an individual in the GA or restrict the test suite size to that of the population size in the GA. DSGA-ST is adaptive in nature and can identify test suites for any size. This allows DSGA-ST to develop test case suites without the limitation of other algorithms.

2. Literature Review

There is a lot of research on Structural Testing. It is one of the earlier forms of testing and has shown to be very effective at finding defects [5]. This is a mature area of research, with decades of cumulative experience, making an exhaustive survey of the area challenging in its own right. Even when considering using GAs for Structural Testing the number of research papers published is very large [6].

Therefore, our discussion only surveys a subset of papers on GAs and a review of related papers on the topic of using GAs for structural testing. This research uses the Triangle Classification algorithm to obtain results, so we employed the Triangular Classification algorithm as a benchmark, to provide a cardinal metric of performance gains, which is readily compared.

2.1. Genetic Algorithms

GAs have proven very useful in solving optimization problems. They have been used for decades to solve problems with large complex domains. Not only have they been used for structural testing, they have been used for countless other problems like predicting stock prices [11], optimization of large complex functions [10] and determining how amino acids fold to create proteins [12].

GAs search through the domain in a similar manner to that via which species evolve to adapt to their environment. A species of plants or animals have a genetic sequence and different values for these genes produce individuals with different traits. The species lives in some environment and in this environment individuals with certain traits do better than other individuals. Individuals that have traits that do well at solving the problem of existing in the environment are more likely to reproduce and pass their traits onto their offspring. Over time the species finds a small set of optimal gene values to cope with the environment [13].

GAs are computer algorithms that model this evolution process that species go through. GAs have individuals which represent a possible solution to the problem. A set of individuals is a generation. The algorithm uses genetic operations on a generation to produce a new generation, which is better at solving the problem than the previous generation. Over many generations, possibly hundreds or thousands, the individuals in the current generation converge to the optimal set of genes and thus to the optimal value in the domain.

GAs implement individuals which represent a possible solution to the problem that the GA is attempting to solve [14]. Individuals are represented as strings, most often comprising 1's and 0's. Different combinations of these 1's and 0's represent different values of the domain. In problems of functional optimization these would represent values for the input parameters to the function. But individuals could model much more complex entities, like classification rules [15-16]. A GA is initialized by randomly generating some number of individuals. This is termed the first generation.

Accompanying the individual is a fitness function. The fitness function accepts an individual as a parameter and returns how well the individual solves the problem. The fitness function is used to determine the best individuals within a generation. Just like in nature the traits of these individuals are more likely to be passed onto future generations.

The GA will perform selection, crossover and mutation iteratively, with each iteration creating the next generation. In the first step of the iteration, selection, the GA selects pairs of individuals for crossover. Individuals with higher fitness are more likely to be selected and individuals can be used in multiple selection pairs. Eventually each pair of individuals from the selection step will produce two new individuals. To keep the number of individuals in each generation consistent, the selection process will produce pairs equal to half of the number of individuals in a generation. Because individuals with higher fitness are more likely to be selected, their genes are more likely to be pass onto the next generation.

The next step is crossover. In the crossover step, a position is randomly selected between the characters in the individual and each individual is split along this position. Switching the tails of the individuals creates two new individuals. These two new individuals become part of the next generation. An example of crossover can be seen in Figure 1.

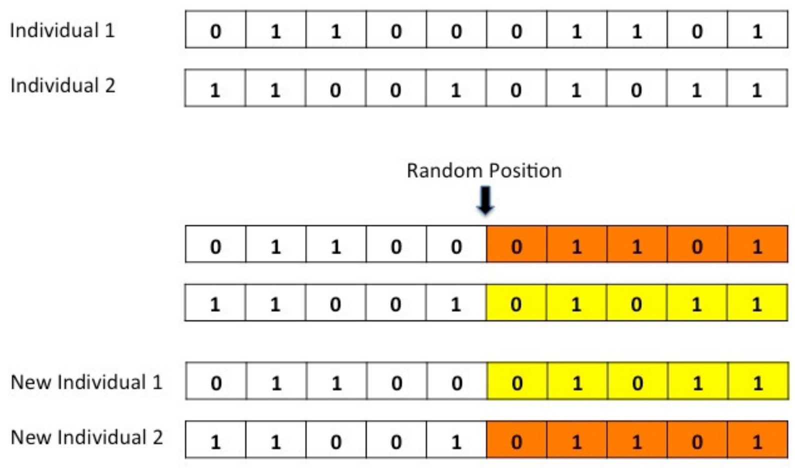


Figure 1. Crossover Example

The final step is mutation. Mutation alters the value of a character based upon a parameter, mutation rate. When an individual has binary characters, the mutation operator just switches the gene value between 0 and 1. When an individual has genes with multiple values another random number is produced to determine the new gene value.

GA theory argues that over many generations the individuals in the population will converge to the correct answer [17]. There are two opposing forces in a GA. Selection and crossover are exploitive forces that work to narrow the generation into optima. Mutation is an explorative force that introduces new areas of the domain for consideration. Over many generations the individuals within the population converge to optima.

2.2. Employing the GA as Structural Test Generator

Because GAs are appropriate for searching, they are well suited for structural test case generation. A significant number of research papers have been published on using GAs for test case generation. We have identified more than a 100 works only on the subject of using GAs for software testing. This review only lists a few relevant ones. We survey only those of specific relevance.

2.2.1. GADGET

Genetic Algorithm Data Generation Tool, GADGET, [18] uses a coverage table to track the areas of the program that test cases have been created for. GADGET creates a specialized objective function for each entry on the coverage table. So if a target condition is if (c >= d), the fitness function would be:

$$\psi(x) = \begin{cases} d - c, if\ d \geq c \\ 0, otherwise \end{cases} \qquad (1)$$

GADGET uses a fitness function which is a model of a minimization problem, so this function would encourage a solution that has c >= d. When test data is found the coverage table is updated with test case values. This opens the possibilities for coverage within the body of the if statement to be located.

Michael, McGraw and Schatz [18] research shows results against a number of programs. Table 1 shows the results for the Triangle Classification algorithm as reported in Sofokleous and Andreou [2]. It produced 9 test cases that have 79.6% coverage over the edges and conditions [2].

A limitation to GADGET is that the GA needs to be run multiple times each time finding a test case. Different fitness functions are derived for the conditions within the target program. Test cases for some areas of the target program cannot be obtained until test cases for previous areas are obtained. This occurs when targets are in the body of conditional statements [2].

Table 1. GADGET results for TC Algorithm as reported in [2]

Test #	**I**	**j**	**k**	**Edges**	**Conditions**
1	1680498885	1961702355	-1490056820	1, 2	A1=F; A2= F; A3=T
2	1293470477	1898197634	465181194	1, 3, 4, 7, 10, 13, 14, 15, 16	A1=F; A2=F; A3=F; B1=F; C1=F; D1=F; E1=T; F1=F; F2=F; F3=T
3	-120192928	1041962067	280365949	1, 2	A1=T
4	841354299	-1802686561	-209782592	1, 2	A1=F; A2=T
5	1056804119	660913846	1617709752	1, 3, 4, 7, 10, 13, 14, 17, 18	A1=F; A2= F; A3=F; B1=F; C1=F; D1=F;

					E1=T; F1=F; F2=F; F3=F
6	719320455	507534636	574028437	1, 3, 4, 7, 10, 13, 14, 17, 18	A1=F; A2=F; A3=F; B1=F; C1=F; D1=F; E1=T; F1=F; F2=F; F3=F
7	743820356	743820356	1826109949	1. 3, 4, 5, 6, 10, 13, 19, 22, 25, 28, 31, 32	A1=F; A2=F; A3=F; B1=T; C1=F; D1=F; E1=F; G1=F; H1=F; I1=T; I2=T
8	999699718	584551117	999699718	1, 3, 4, 7, 8, 9, 13, 19, 22, 25, 26, 27	A1=F; A2=F; A3=F; B1=F; C1=T; D1=F; E1=f; G1=F; H1=F; I1=T; I2=T
9	799340978	1321708382	1321708382	1, 3, 4, 7, 10, 11, 12, 19, 22, 28, 29, 30	A1=F; A2=F; A3=F; B1=F; C1=F; D1=T; E1=F; G1=F; H1=F; I1=F; J1=T; J2=T

2.2.2. Automatic Test Cases Generation System

The Sofokleous and Andreou [2] algorithm comprises of two sub-systems. The Basic Program Analyzer System (BPAS) analyzes the program, produces a Control Flow Graph and determines code coverage. The Automatic Test Case Generation System (ATCGS) searches the input space for an optimal set of test cases for structural coverage [2]. ATCGS comprises of two algorithms: Batch-Optimistic (BO) and Close-Up (CU).

The BO employs a variant of McCabe's Cyclomatic Complexity formula to determine the number of test cases needed for coverage. A complete set of test cases is then modeled as an individual for the GA. The BO employs the following fitness function:

$$F = \frac{w_1(\#edges_{executed}) + w_2(\#pred_{true} + \#pred_{false})}{w_1 + w_2} \qquad (2)$$

In this fitness function w1 and w2 are weights. #edgesexecuted is the number of edges executed by the test cases and #predtrue and #predfalse are the number of predicates evaluated to true and false respectively. BO stores these test cases in a repository.

The CU algorithm attempts to find test cases for other conditions in the target program not located by BO. It creates a GA for each condition and where the GA cannot locate a suitable test case, it marks the target area as unreachable. Table 2 shows the results of the Sofokleous and Andreou algorithm [2] on the Triangle Classification algorithm. It produced 10 test cases and provided 100% coverage over the edges and conditions.

Table 2. Automatic Test Cases Generation System results for TC algorithm [2]

Test #	I	J	K	Edges	Conditions
1	93	-1	0	1, 2	A1=F; A2= T
2	-8	-1	-7	1, 2	A1=T
3	5	784	-1	1, 2	A1=F; A2= F; A3=T
4	786	732	1	1, 3, 4, 7, 10,	A1=F; A2= F; A3=F;

				13, 14, 15, 16	B1=F; C1=F; D1=F; E1=T; F1=F; F2=T
5	476	645	537	1, 3, 4, 7, 10, 13, 14, 17, 18,	A1=F; A2= F; A3=F; B1=F; C1=F; D1=F; E1=F; F1=F; F2=T
6	1	1	45	1, 3, 4, 5, 6, 10, 13, 19, 22, 25, 28, 31, 32	A1=F; A2= F; A3=F; B1=T; C1=F1; D1=F; E1=F; G1=F; H1=T; H2=F; I1=F; J1=F
7	1	1	1	1, 3, 4, 5, 6, 8, 9, 11, 12, 19, 20, 21	A1=F; A2=F; A3=F; B1=T; C1=T; D1=T; E1=F; G1=T
8	346	1	597	1, 3, 4, 7, 10, 13, 14, 15, 16,	A1=F; A2=F; A3=F; B1=F; C1=F; D1=T; E1=F; G1=F; H1=F; I1=F; J1=T; J2=T
9	5	881	5	1, 3, 4, 5, 6, 10, 13, 19, 22, 25, 28, 31, 32	A1=F; A2=F; A3=F; B1=F; C1=F; D1=T; E1=F; G1=F; H1=F; H2=F; I1=T; I2=F; J1=F
10	1	6	6	1, 3, 4, 7, 10, 11, 12, 19, 22, 25, 28, 29, 30	A1=F; A2=F; A3=F; B1=F; C1=F; D1=T; E1=F; G1=F; H1=F; I1=F; J1=T; J2=T
11	31	4	31	1, 3, 4, 7, 8, 9, 13, 19, 22, 25, 26, 27	A1=F; A2=F; A3=F; B1=F; C1=T; D1=F; E1=F; G1=F; H1=F; I1=T; I2=T
12	15	15	2	1, 3, 4, 5, 6, 10, 13, 19, 22, 23, 24	A1=F; A2=F; A3=F; B1=T; C1=F; D1=F; E1=F; G1=F; H1=T; H2=T
13	4	1	1	1, 3, 4, 7, 10, 11, 12, 19, 22, 25, 28, 31, 32	A1=F; A2=F; A3=F; B1=F; C1=F; D1=T; E1=F; G1=F; H1=F; I1=F; J1=T; J2=F
14	36	42	5	1, 3, 4, 7, 10, 13, 14, 15, 16	A1=F; A2=F; A3=F; B1=F; C1=F; D1=F; E1=T; F1=F; F2=F; F3=T

2.3. Triangle Classification Algorithm

The Triangle Classification algorithm is a short algorithm that determines if three lengths would form a triangle and, if so, would the triangle be equilateral, isosceles or scalene. The algorithm accepts three parameters, i, j and k, which are the length of the three sides. The algorithm returns an integer: 1 for a scalene, 2 for an isosceles, 3 for an equilateral and 4 for not a triangle. The algorithm is often used in structural testing because it is short, but it has many paths and conditions. Figure 2 shows a control flow graph of the algorithm. Conditional statements are indicated with a letter. If it is a compound condition, each part of the condition is given a number. These are shown in the control flow graph in brackets [ ]. For example [A1] indicates the conditional statement A and the first Boolean expression within the conditional statement. These annotations are used in the results section.

The Triangle Classification algorithm is novel for structural testing for another reason. Assuming that the input parameters can be negative, a vast majority of the possible test case inputs lead to a single path in the control flow. No triangle can have a side of negative or 0 length. Condition A of the control flow checks to see if i, j or k is less than or equal to 0. If so, the three sides cannot make a triangle and it follows path 2 and terminates. If i, j and k have a sign bit, then a vast majority of input data follow this, not so interesting, path through the program. Algorithms that attempt to generate structural test cases need to explore these other areas of the search space.
The Triangle Classification algorithm is a benchmark algorithm for structural testing. In addition to GADGET [18] and Automatic Test Cases Generation System [2] it has been used in other software testing research. Ammann and Offutt argue in [20] that it is a good algorithm for software testing research because it is easy to understand, relatively small yet very complex in terms of paths. In 2012 it was used in [21] for research in mutation testing. It has been used in Master's Thesis [22] and Doctoral Dissertations [23]. It has been used in ant colony optimization algorithms [24] and other research [25]. For these reasons the Triangle Classification algorithm was used in this research.

3. METHODOLOGY

GAs are not always effective at finding optima in multi-optima problems. They can often converge to local optima, and fail to identify global optima [26-27]. One area of GA research is called Niche Genetic Algorithms (NGA). NGAs are specialized GAs that find optima in multi-optima domains [26-27]. These algorithms concentrate on conserving areas of the domain from the exploitative forces of selection and crossover. This allows the GA to explore these weak areas of the domain and possibly find local optima.

This research uses the proto-type research methodology and generates a structurally complete set of test cases for the Triangle Classification algorithm. The results of the program are compared to existing literature on generating test cases for the Triangle Classification algorithm. The DSGA-ST algorithm is used with individuals modeling the input parameters for the Triangle Classification algorithm. Each individual represents a single set of input values.

Our DSGA-ST algorithm produces a list called the tabu list, which contains locally optimal values. In DSGA-ST the tabu list often contains duplicate or redundant values. So, the tabu list is filtered through a short algorithm that removes redundant values. These redundant values could be identical individuals having the same i, j and k values. Or, they could be different individuals, but that follow the same paths through the program. Figure 3 shows a system level diagram of how the test cases are generated. When the redundant test cases are removed, the final list of test cases is the test suite.

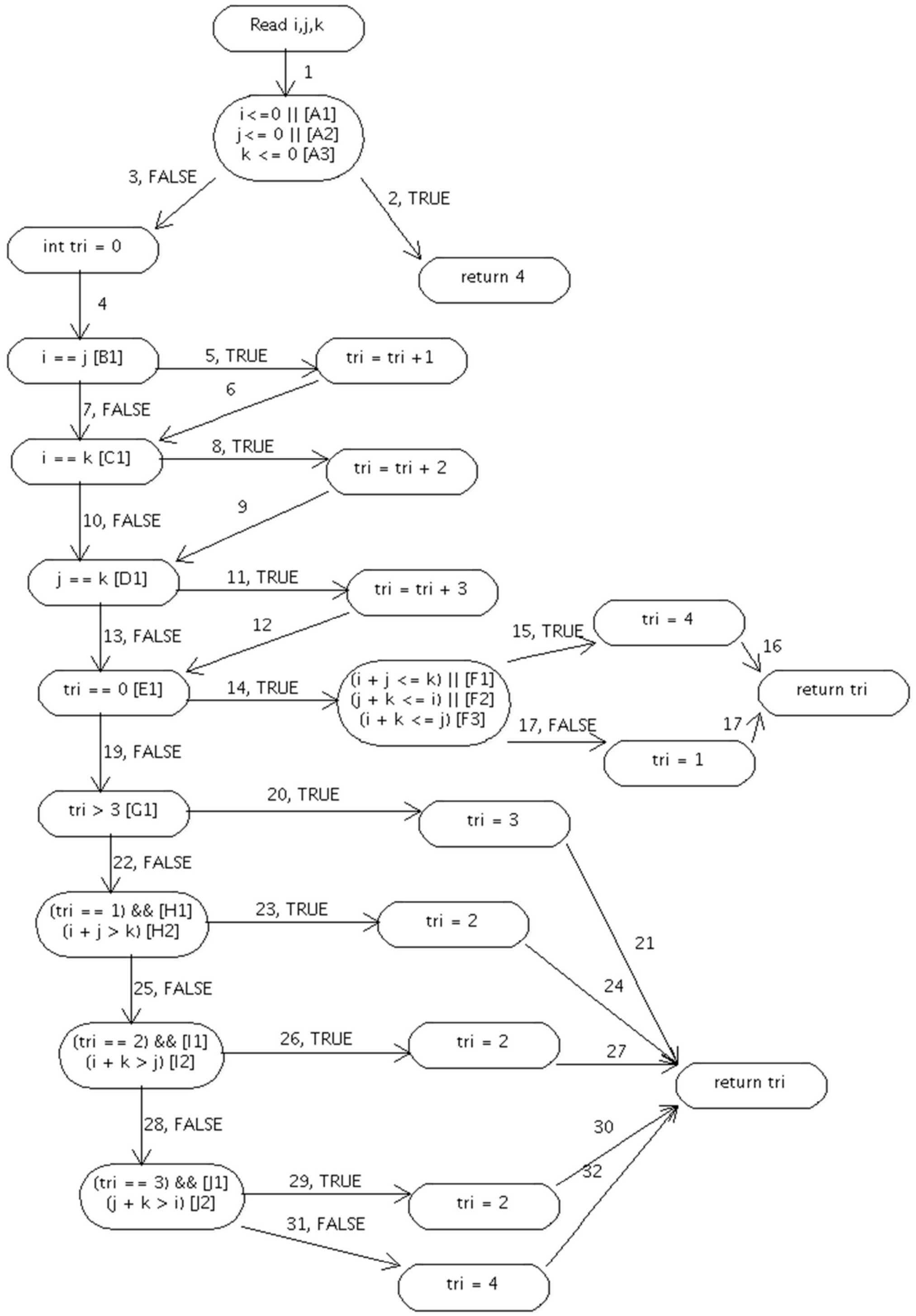


Figure 2. Control Flow Graph of Triangle Classification Algorithm

Notably, the DSGA [10] is a clustering algorithm framework that uses a tabu list and a radius. The tabu list stores possible local and global optima and is used to encourage exploration in other areas of the domain. The radius is used to identify sub-areas of the domain so the algorithm can conserve individuals in these areas. The selection process uses a shared fitness function and tabu list to encourage exploration in other areas of the domain. DSGA changes its radius as the algorithm runs. Varying the radius in combination with the tabu list compensates for poor choices in radius values, which hinder other NGAs.

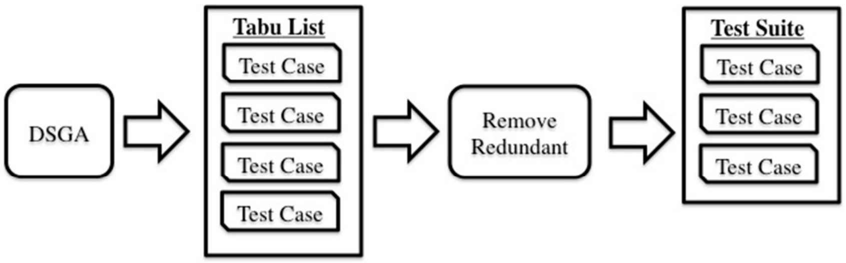


Figure 3. System Level Diagram of Algorithm

3.1. Algorithm Overview

As with any GA, individuals need to model potential domain values of the problem. To generate a set of test cases the algorithm models the three input parameters to the Triangle Classification algorithm, where each represents the length of a side of the triangle. However, the Triangle Classification algorithm does handle input parameters that cannot form a valid triangle. So, these other values are possible inputs to the algorithm.

In this research the three sides to the triangle are represented as 8-bit integers with a 9th-bit used for the sign. Since there are three parameters the total length of an individual is 27 bits. Bytes were transferred using the little endian representation.

In DSGA [10] a distance measurement needs to be defined between two individuals. In a GA there are many ways to represent distance. There is Euclidian distance, chromosomal distance and possibly more ways. In the Triangle Classification algorithm there are a total of 48 edges and conditions. In this algorithm distance is computed by summing the total number of edges and conditions that two individuals have in common. The distance is 48 divided by this sum. So two individuals are considered close together if they share many of the same edges and conditions. They are considered far apart if they have many different edges and conditions. This distance measurement will encourage exploration by finding other test cases that explore other parts of the code.

Table 3 shows the edges and conditions executed in the Triangle Classification algorithm for examples of (0, 1, 1) and (1, 1, 1). These two individuals have only one edge in common, Edge 1. They have three conditions in common, A1, A2 and A3. That is a total of 4 edges and conditions. The distance between (0, 1, 1) and (1, 1, 1) is 48 / 4 = 12. However, if we look at (2, 2, 2) it would follow the same path through the graph as (1, 1, 1). So, (1, 1, 1) and (2, 2, 2) would have all 20 edges and conditions the same and thus would have a distance of 48 / 20 = 2.4. Even though (0, 1, 1) and (1, 1, 1) share two of the three values, they are far apart in terms of coverage of the control flow graph. However, (1, 1, 1) and (2, 2, 2) share no values in common but are identical in the control flow graph.

Table 3. Edges (E) and Conditions for examples (0, 1, 1) and (1, 1, 1)

i = 0
j = 1
k = 1
	Edge 1, A1, A2, A3, Edge 2

i = 1
j = 1
k = 1
	Edge 1, A1, A2, A3, Edge 3, Edge 4, B1, Edge 5, Edge 6, C1, Edge 8, Edge 9, D1, Edge 11, Edge 12, E1, Edge 19, G1, Edge 20, Edge 21

3.2. Fitness Function

With any GA the fitness function is a function that the GA will find optimal values for. So, the fitness function should model the problem trying to be solved. In many Multi-optima GAs, like DSGA, there are two fitness functions. The Natural Fitness is the fitness function that describes the problem to be solved. The Shared Fitness function [26] will adjust an individual's Natural Fitness to encourage exploration in other areas of the domain. DSGA [10] accomplishes this by decreasing the fitness of individuals the closer that they are to individuals on the tabu list. This is due to the tabu list already containing possible optimal values. Further exploration around tabu list values is not helpful. The Shared Fitness encourages exploration in other areas of the domain. In this research the Natural Fitness is defined as the sum of the number of edges and conditions covered by the individual. This is done because a goal of structural testing is to find a smallest set of test cases that cover the structure of the program. The Shared Fitness is defined as the Natural Fitness divided by mi where mi is defined as the following:

$$mi = \sum_{x=1}^{\substack{Tabu \\ List\ Length}} MaxDistance - \frac{Distance(i, TabuList(x))}{MaxDistance * 0.1} \qquad (3)$$

In equation 3 Tabu List Length is the number of individuals on the tabu list. MaxDistance is the maximum distance between two individuals. In the Triangle Classification algorithm this value is 48. TabuList(x) is the xth individual on the tabu list. Defining mi in this way ensures that as an individual i gets closer to individuals on the tabu list mi will increase and the Shared Fitness will decrease.

When DSGA-ST is initialized the tabu list is empty. This makes it difficult to compute Shared Fitness. So, Shared Fitness is simply the Natural Fitness until individuals are added to the tabu list.

3.3. DSGA Algorithm – Low Level

DSGA enhances a traditional GA with Seed Selection and Seed Conservation as defined in the SCGA algorithm [28]. A Seed is defined as a locally strong individual. For each generation of the GA the individuals are evaluated in order of fitness. An individual is marked as a seed if no other seed with a radius parameter, r, is a seed. Since the individuals are evaluated in order of fitness this ensures that the locally strong individuals are seeds. In the example shown in Figure 4 the seed selection step would evaluate the individuals in the following order: 3, 2, 4, 1, 5, 6. This is the order of their fitness. It is obvious that individual 3 is a seed. Since individuals 1, 2, 4 and 5 are within the radius of individual 3, they are not designated as seeds. Even though individual 6 is relatively weak, it is outside of the radius of individual 3. When it is evaluated there are no individuals within the radius that are seeds, so it is designated a seed. In this example individuals 3 and 6 are seeds.

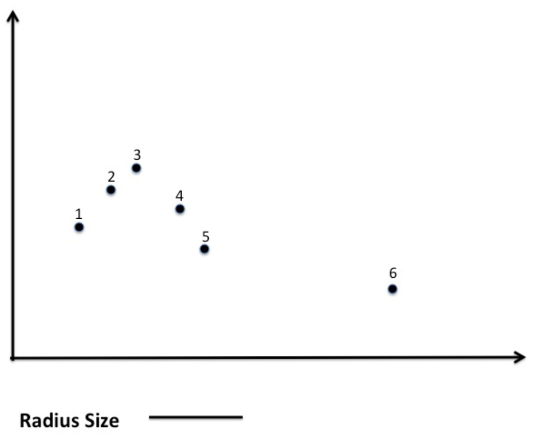


Figure 4. Seed Selection Example

Once seeds are determined DSGA goes through the traditional GA steps of Selection, Crossover and Mutation. The final step in the GA loop is Seed Conservation. Each seed will replace an individual in the next generation, thus ensuring their genetic traits are passed on. The Seed Conservation [28] step takes each seed and replaces the weakest individual within the radius, r, of the seed in the next generation. If no individuals are within the radius, r, of the seed in the next generation, then the seed replaces the globally weakest seed in the next generation. Once in the next generation a seed loses its seed designation and needs to go through the Seed Selection process again to determine if it is a seed. Reference the DSGA pseudo-code for DSGA in Table 4. Periodically DSGA executes a new step called Reevaluation, which looks for strong areas of the domain. These areas are placed on the Tabu List. A parameter called Reevaluation Loop Count, RLC, indicates the number of generations between each Reevaluation step. Individuals can be placed on the Tabu List in two different ways. The current seeds are placed on the Tabu List. The Reevaluation step also looks for areas in the population that show convergence. A parameter called Convergence Limit, CL, is used as a threshold. If in the current generation during Reevaluation there are CL or more identical individuals then one of the individuals is placed on the Tabu list. All individuals in the current generation that were placed on the Tabu List are replaced with randomly generated individuals.

Table 4 shows the DSGA pseudocode. DSGA has shown promise in a number of areas of search and optimization. In [10] DSGA was used to solve a series of functional optimization problems. Later in Brown, Pelosi and Dirska [11] DSGA was used to predict stock prices. While selecting stocks from the Dow Jones Industrial Index the stocks selected outperformed the underlying index by nearly 400%. Recently it has been used in de novo protein folding problem [12]. Research shows that DSGA is a very good optimization algorithm.

Table 4. DSGA pseudocode [10]

Line #	**Pseudocode**
1	Initialization
2	While not termination condition
3	For (int r = 1; r < RLC; r++)
4	Seed Selection
5	Selection

6	Crossover
7	Mutation
8	Seed Conservation
9	End for loop
10	If there exists an individual d with CL or more identical individuals then
11	Add d to tabu_list
12	Replace all individuals identical to d with randomly generated individuals
13	End if
14	Add the seeds of the current generation to the tabu_list
15	Replace all individuals that are seeds with randomly generated individuals
16	Alter radius by SD
17	End while loop

4. RESULTS

We executed the DSGA-ST algorithm with removals of redundancy against the Triangle Classification algorithm using the parameters shown in Table 5. The individuals, which represent the three input values for the test cases, were 27-characters long with each character being a 0 or a 1. There are 1.34 x 108 unique combinations of values that an individual can have. With these parameters only 200,000 values are actually evaluated. This is less than 0.2% of the total unique combination of values.

With the parameter settings shown in Table 5, DSGA-ST can produce a suite of tests cases that cover each statement and all values, TRUE or FALSE, for each condition. The set of test cases is shown in Table 6 along with the statements covered and values for each of the conditions. This suite of tests cases executes each statement and produces at least one TRUE and FALSE for each condition.

Table 5. Parameter values for DSGA-ST

Parameter	**Description**	**Value**
Population Size	The number of individuals per generation	200
Number of Generations	The number of generations before termination	1,000
Mutation Rate	The probability that a gene will mutate	0.03
Initial radius	The initial value of the radius	1.0
Radius delta	The amount that the radius is changed	0.1
Reevaluation loop count	The number of generations needed to evaluate individuals for the Tabu list	25
Convergence limit	The number of identical individuals to conclude that convergence has taken place	2

Table 6. DSGA Triangle Classification Test Suite

Test #	**i**	**j**	**k**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
1	-117	200	96	X	X												
2	32	224	64	X		X	X			X			X			X	X
3	40	40	62	X		X	X	X	X				X			X	
4	64	218	64	X		X	X			X	X	X				X	

5	72	-229	-23	X	X												
6	75	16	16	X		X	X			X			X	X	X		
7	90	90	184	X		X	X	X	X				X			X	
8	96	32	224	X		X	X			X			X			X	X
9	161	142	140	X		X	X			X			X			X	X
10	196	120	60	X		X	X			X			X			X	X
11	223	174	174	X		X	X			X			X	X	X		
12	224	224	224	X		X	X	X	X		X	X		X	X		
13	226	88	-89	X	X												
14	252	162	252	X		X	X			X	X	X				X	

Table 6 (Continuation). DSGA Triangle Classification Test Suite

Test #	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
1																
2	X	X														
3					X			X	X	X						
4					X			X			X			X		
5																
6					X			X			X			X		
7					X			X			X			X		
8	X	X														
9			X	X												
10	X	X														
11					X			X			X			X	X	X
12					X	X	X									
13																
14					X			X			X	X	X			

Table 6 (Continuation). DSGA Triangle Classification Test Suite

Test #	31	32	A1	A2	A3	B1	C1	D1	E1	F1	F2	F3	G1	H1	H2	I1	I2
1			T	F	F												
2			F	F	F	F	F	F	T	F	F	T					
3			F	F	F	T	F	F	F				F	T	T	F	T
4	X	X	F	F	F	F	T	F	F				F	F	T	T	F
5			F	T	T												
6	X	X	F	F	F	F	F	T	F				F	F	T	F	T
7	X	X	F	F	F	T	F	F	F				F	T	F	F	T
8			F	F	F	F	F	F	T	T	F	F					
9			F	F	F	F	F	F	T	F	F	F					
10			F	F	F	F	F	F	T	F	T	F					
11			F	F	F	F	F	T	F				F	F	T	F	T
12			F	F	F	T	T	T	F				T	F	T	F	T
13			F	F	T												
14			F	F	F	F	T	F	F				F	F	T	T	T

Table 6 (Continuation). DSGA Triangle Classification Test Suite

Test #	J1	J2
1		
2		
3	F	T
4	F	T
5		
6	T	F
7	F	T
8		
9		
10		
11	T	T
12	F	T
13		
14	F	T

4.1. Discussion of Results

The results show that DSGA is very effective at generating structurally complete test cases. In only 14 test cases it produced a structurally complete test suite. This is equally as good as the ATCGS algorithm and better than the GADGET algorithm. A set of 14 test cases is still relatively small considering the complexity of the algorithm.

Both ATCGS and DSGA-ST produced the same number of test cases at 14. However, in ATCGS the entire set of test cases is modeled with each individual. For larger programs this could be an issue. DSGA-ST models each test case as an individual in the GA and the set of test cases comes from the tabu list with can have an unlimited size. This makes DSGA-ST more flexible.

4. CONCLUSION

This research introduces a new algorithm for automated generation of structural test cases. The algorithm uses a specialized GA to locate some optimal values and stores them in a short-term memory structure. As these optimal areas of the domain are discovered, the algorithm encourages exploration in other areas of the domain. When the GA finishes a step is performed to remove redundant values, see Figure 3. The remaining test cases make up a complete test suite.

DSGA-ST does not perform any analysis on the target program. It does not make specialized fitness functions for specific areas of the program. Other than determining which areas of the target program are covered by a test case, it is completely free of target specific constraints. This can allow DSGA-ST to generate test cases on much larger programs than the Triangle Classification algorithm.

Through the use of DSGA-ST and the fitness function, this algorithm generates test cases the way that many humans would do it by hand. The Natural Fitness function is defined as the sum of the number of edges and conditions that the test case covers. This encourages the algorithm to find test cases that cover large areas of the Triangle Classification algorithm. Once some good test cases are found, the algorithm looks in other areas to find more good test cases.

Another benefit that DSGA-ST has over other algorithms is that the number of test cases needed does not have to be known prior to using the algorithm. In many algorithms the number of test cases produced cannot exceed the population size of the GA. In DSGA-ST the test cases are derived from the tabu list which can be larger than the population size.

A future area of this research is to apply DSGA-ST to larger programs and conduct an empirical study in a software testing organization. This future research should compare defect rates between software tested through DSGA-ST and other testing methods within the same organization.

When generating structurally complete test cases there are two objectives. The primary objective is to cover the structure of the program. But a secondary objective is to minimize the number of test cases. In most GAs all objectives are modeled within the fitness function. But in DSGA-ST the fitness function only models the number of parts of the program that are covered by a test case, which is the secondary objective. The primary objective is handled through the tabu list, distance measurement and the shared fitness. This shows the effectiveness of these components of the algorithm.

DSGA-ST is a new approach to generating structural test cases. Enhancing a GA with a tabu list, shared fitness and seeds makes DSGA-ST suitable for finding test cases of large complex programs. DSGA-ST has a number of advantages over existing algorithms. Future research may show that it is advantageous in other ways.

REFERENCES

[1] S. Anand, E. Burke, T. Y. Chen, J. Clark, M. B. Cohen, M. B. et. al., "An Orchestrated Survey of Methodologies for Automated Software Test Case Generation", Journal of Systems and Software 86(8), pp. 1978-2001, 2013.

[2] A. A. Sofokelous & A. S. Andreou, A. S., "Automatic, evolutionary test data generation for dynamic software testing", The Journal of Systems and Software 81, pp 1883-1898, 2088.

[3] N. K. Gupta & M. K. Rohil, "Using Genetic Algorithm for Unit Testing of Object Oriented Software", Proceedings of the First International Conference on Emerging Trends in Engineering and Technology, pp. 308-313, 2008.

[4] J. Wegener & O. Bühler, "Evaluation of different fitness functions for the evolutionary testing of an autonomous parking system", In Genetic and Evolutionary Computation–GECCO 2004, pp. 1400-1412, 2004.

[5] M. Pezze & M. Young, Software testing and analysis: process, principles, and techniques. John Wiley & Sons, 2008.

[6] M. S. Brown, F. Neptune, G. L. Bohl, M. A. Cifuentes, J. Young, et al., "A Survey of the use of Genetic Algorithms in Structural Testing", International Journal of Advanced Research in Computer Science and Software Engineering, 3(12), pp. 6-13, 2013.

[7] S. Nachiyappan, A. Vimaladevi, & C. B. SelvaLakshmi, "An Evolutionary Algorithm for Regression Test Suite Reduction", Proceedings of the International Conference on Communication and Computational Intelligence, pp. 503-508, 2010.

[8] S. Khor & P. Grogono, "Using a genetic algorithm and formal concept analysis to generate branch coverage test data automatically", Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 346-349, 2004.

[9] W. Weimer, T. V. Nguyen, C. Le Goues & S. Forrest, "Automatically finding patches using genetic programming", International Conference on Software Engineering, Vancouver, Canada, pp. 364-374, 2009.

[10] M. S. Brown, (2010). "A Species-Conserving Genetic Algorithm for Multimodal Optimization", PhD. Dissertation, Nova Southeastern University, Fort Lauderdale, FL, 2010.

[11] M. S. Brown, M. Pelosi & H. Dirska, "Dynamic-radius Species-conserving Genetic Algorithm for the Financial Forecasting of Dow Jones Index Stocks", Machine Learning and Data Mining in Pattern Recognition, 7988, pp. 27-41, 2013.

[12] M. S. Brown, T. Bennett & J. A. Coker, "Niche Genetic Algorithms are better than traditional Genetic Algorithms for de novo Protein Folding", F1000 Research, 3(236) (doi: 10.12688/f1000research.5412.1), 2014.

[13] C. Darwin, On the origin of species by means of natural selection. London: Murray, 1859.

[14] H. J. Bremermann, "The Evolution of Intelligence: The Nervous System as a Model of its Environment", Technical Report, No.1, Contract No. 477, Issue 17. Seattle WA: Department of Mathematics, University of Washington, 1958.

[15] K. A. De Jong, W. M. Spears & D. F. Gordon, D. F. "Using Genetic Algorithms for Concept Learning", Machine Learning 13(2-3), pp. 161–188, 1993.

[16] L. B. Booker, D. E. Goldberg & J. H. Holland, "Classifier Systems and Genetic Algorithms", Artificial Intelligence 40, pp. 235-282, 1989.

[17] J. H. Holland, Adaptation in Natural and Artificial Systems. Ann Arbor, MI: University of Michigan Press, 1975.

[18] C. C. Michael, G. McGraw & M. A. Schatz, "Generating software test data by evolution", IEEE Transactions on Software Engineering 27(12), pp. 1085–1110, 2001.

[19] T. J. McCabe, "A complexity measure", IEEE Transactions on Software Engineering SE-2(4), pp. 308-320, 1976.

[20] P. Ammann & J. Offutt, Introduction to Software Testing. Cambridge University Press, Cambridge, UK, 2008.

[21] V. H. S. Durelli, J. Offutt & M. E. Delamaro, "Toward Harnessing High-Level language Virtual Machines for Further Speeding up Weak Mutation Testing", 2012 IEEE Fifth International Conference on Software Testing. Verification and Validation, pp. 681-690, 2012.

[22] A. Panchapakesan, A Hybrid Genetic Algorithm and Evolutionary Strategy to Automatically Generate Test Data for Dynamic, White-Box Testing, Master's thesis, University of Ottawa, 2013.

[23] V. H. S. Durelli, Toward harnessing a Java high-level language virtual machine for supporting software testing, PhD dissertation, Universidade de São Paulo, 2013.

[24] S. Yang, T. Man, & J. X. "Improved Ant Algorithms for Software Testing Cases Generation", The Scientific World Journal, 2014.

[25] P. McMinn, "Search based software test data generation: a survey", Software testing, Verification and reliability, 14(2), pp. 105-156, 2004.

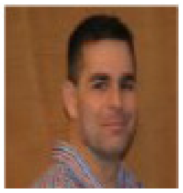
[26] D. E. Goldberg & J. Richardson, "Genetic Algorithms with Sharing for Multimodal Function Optimization", Proceedings of the Second International Conference on Genetic Algorithms and their Application, Cambridge Massachusetts, pp. 41-49, 1987.

[27] K. A. De Jong, An analysis of the behavior of a class of genetic adaptive systems, PhD. Dissertation, University of Michigan, 1975.

[28] J. P. Li, M. E. Balazs, G. T. Parks & P. J. Clarkson, "A Species Conserving Genetic Algorithm for Multimodal Function Optimization", Evolutionary Computation, 10(3), pp. 207-234, 2002.

AUTHORS

Dr. Michael Brown spent nearly 20 years working in the Information Technology field. Currently he is the Program Chair at University of Maryland University College for the Software Engineering Master's degree in Information Technology. He has work for some well-known organizations like Sun Microsystems and NASA.



Dr. Michael Pelois is a faculty member at the University of Maryland University College and East Central University. He has published numerous papers in the field of artificial intelligence and the defence industry.