Dynamic White-Box Software Testing using a Recursive Hybrid Evolutionary Strategy/Genetic Algorithm

Ashwin Panchapakesan*, *Graduate Student Member*, Rami Abielmona†, *Senior Member, IEEE*, and Emil Petriu‡, *Fellow, IEEE*
*School of EECS, University of Ottawa, Ottawa, Canada
†Research & Engineering, Larus Technologies Corporation, Ottawa, Canada
‡School of EECS, University of Ottawa, Ottawa, Canada
*apanc006@uottawa.ca, †rami.abielmona@larus.com, ‡petriu@eecs.uottawa.ca

***Abstract*—Software testing is an important and time consuming part of the software development cycle. While automated testing frameworks do help in reducing the amount of programmer time that testing requires, the onus is still upon the programmer to provide such a framework with the inputs on which the software must be tested. This requires static analysis of the source code, which is more effective when performed as a peer review exercise and is highly dependent on the skills of the programmers performing the analysis. Thus, it demands the allocation of precious time of highly skilled programmers. An algorithm that automatically generates inputs to satisfy test coverage criteria for the software being tested would therefore be valuable, as it would imply that the programmer no longer needs to analyze code to generate the relevant test cases. This paper explores a hybrid evolutionary strategy with an evolutionary algorithm to discover such test case synthesis, in an improvement over previous methods which overly focus their search without maintaining the diversity required to cover the entire search space efficiently.**

Keywords*—Software testing, black-box testing, white-box testing, static white-box testing, dynamic white-box testing, evolutionary algorithm, genetic algorithm, evolutionary strategy, control flow graph

I. Introduction

A. The Need for Software Testing

As part of the development process, software needs to be tested before it is deployed. This is becoming more and more relevant in today's world, given how many critical systems are controlled by software. A bug in mission critical software could have severe consequences, including the loss of human life. As a result, it is imperative that software be thoroughly tested before it is deployed. As such, the testing process consumes approximately 50% of the software development timeline [1]. Considering how expensive it is to develop software, it is clear that reducing the amount of time spent on testing the SUT (Software Under Test) would lead to an optimization of the time spent in the development process and therefore an optimization in the cost of producing software. This reduction in time may be achieved by automating the process of testing.

Testing the SUT requires that the agent performing the tests (whether that agent is human or a computer program) provides the software with some inputs and compares the observed behavior against the expected behavior. If the SUT behaves as expected, it passes the test, else it fails the test. The running of the tests themselves can be automated by software in the form of test scripts which can be reused several times if developed correctly [2]. It is the generation of input data that makes up the test cases, which seems to require human input and resources. It is the generation of such test case data that is of interest in this paper and will be explained in the sections to follow.

The rest of the paper is structured as follows: Sections I-B, I-C introduce some software testing paradigms and Section I-D introduces the two evolutionary paradigms that comprise the proposed hybrid. While section II presents some previous work in the field, section III describes how these paradigms are used to create the recursive hybrid evolutionary algorithm and results are presented in Section IV with some discussion in Section V.

B. Software Testing Methodology Overview

Software testing can be broadly categorized into static and dynamic testing, explained in the following subsections.

1) Static Software Testing: Under the static testing paradigm, code reviewer(s) perform code reviews and walk-throughs of the SUT with hypothetical inputs, visually following the logical program flow. This is highly dependent on the skill of the reviewer and requires a lot of the reviewer's time [3, 4].

Further improvements in static testing allowed code to be symbolically analyzed, collecting predicates for the various paths of execution of the code. From these predicates, it is determined which paths may be infeasible or non-executable [5]. Others have used such an approach and combined these predicates with a constraint solver to determine which paths may be infeasible in a SUT [6].

2) Dynamic Software Testing: On the other hand, under the dynamic testing paradigm, the code for the SUT is actually run with the given test inputs. The behavior of the SUT is observed and compared against its expected behavior and the test passes

or fails depending on whether the observed behavior matches the expected behavior.

As outlined in [7], dynamic testing can be split into two categories:

1) **Black box testing**: also known as functional testing; tests the SUT to ensure that it is faithful to the specifications from which it was authored
2) **White box testing**: also known as structural testing; tests the SUT to attain some level of code coverage and to test it on boundary conditions, etc.

C. Dynamic White Box Testing

White box testing is a software testing paradigm that uses the source code of the SUT to test it. It is used to ensure that all parts of the code's structure are executable, i.e. to ensure code coverage. As such, there are several forms of white-box testing. In each form, the SUT is converted into a control flow graph (CFG) - a mathematical representation of the logical program flow of the SUT. In a CFG, each statement is a node and sequential statements are connected by edges. Branching statements (`if-then-else` statements, `for-loops` and `while-loops`) are characterized by multiple edges emanating from a single node, with conditions on each edge.

The core principle around dynamic, white-box testing is that some code coverage criteria must be met. Given a CFG of the SUT, there are many notions of coverage (node coverage, edge coverage, etc). Among these coverage criteria, path coverage is the strongest testing criterion [3], and it is widely accepted as a natural criterion of program testing completeness [5]. It requires that every path in the CFG (from the source node to any terminal node) be executed at least once by the test suite.

D. Evolutionary Algorithms

1) Genetic Algorithms: As outlined by Juang [8], a GA (Genetic Algorithm) encodes a candidate solution to a problem in "individuals" (or "chromosomes") of a "population". After initializing a random population of individuals, the latter are evaluated for fitness (a measure determining the optimality of the candidate solution in its ability to solve the problem). Members of the current generation of the population are selected in a manner proportional to their fitness, to undergo crossover and mutation operations so that parts of these individuals may be combined with each other in the hopes of creating a better individual for the next generation. Through several iterations of such selection, crossover, and mutation operations, the next generation of the population is created. Several iterations of creating such next generations are performed, in the hopes of eventually finding an individual with an acceptable fitness value.

The various components of a GA are explained in the following subsections:

Chromosome: A chromosome is an encoding of a candidate solution to the given problem. In the context of discovering useful test input data, a chromosome may encode an input vector (i.e. a vector of values - one per input variable in the SUT) [9].

Individual: An individual is a full representation of a candidate solution to the problem. Therefore an individual may be a single chromosome or a collection of chromosomes. Since the individuals in the GA used in this paper have only one chromosome, the terms "chromosome" and "individual" are used interchangeably.

Population Generation: In order to begin the process of evolution, the first generation of the population needs to be synthesized to cover the search space as evenly as possible, so that the subsequent genetic operations (crossover and mutation operations) do not get stuck in local optima. This promotes genetic diversity so that some mating operations would allow the GA to escape local optima. This is achieved by creating a random initial population. Thus, for a population of bit-string chromosomes, *n* bits long, each chromosome is generated by concatenating the results of *n* calls to a random bit generator [9].

Fitness: In order for a GA to evolve better individuals over time, the measure of the quality of each individual is provided by an objective fitness function. Since each individual encodes an input vector for the SUT, the fitness function captures differences between the individual-induced path and the target path, as explained in Section III-D.

Selection: In order to produce better individuals for the next generation of the population, appropriate individuals must be selected from the current population. Intuitively, a selection mechanism would favor fitter individuals over unfit ones. To that end, a biased roulette wheel selection mechanism is used.

A biased roulette wheel selection models a situation where individuals bet on a roulette wheel, and the winner will be chosen for genetic (crossover and mutation) operations [10, 11]. In order to bias the bets that the individuals place, based on their fitnesses, each individual is to bet on an entire section of the roulette wheel. The size of the section is directly proportional to the fitness of the individual, relative to the population [5]. For example, if a population consists of three individuals whose fitness scores are as shown in Table I

Table I: Fitnesses of the Individuals in a Population

Individual	Raw Fitness	Relative Fitness [1]
1	5	0.5
2	1	0.1
3	4	0.4

And the biased roulette wheel was modeled as the continuous interval $[0, 1]$, then, the individuals are allowed to bet on sections of the roulette wheel as shown in Table II

Table II: A Biased Roulette Wheel

Individual	**Section of the Roulette Wheel**
1	0.0 - 0.5
2	0.5 - 0.6
3	0.6 - 1.0

[1] Relative fitness is computed as $\frac{fitness(p)}{\sum_{p \in population} fitness(p)}$

A selection operation is performed by picking a random number in the interval and selecting the individual within whose block that number falls. Thus if the random number was 0.506934, then *individual 2* is selected for mating operations.

Crossover: In order to combine components of known candidate solutions to form new candidate solutions, a process called crossover is employed. In the process of crossover, parts of corresponding pairs of chromosomes from two parent individuals are recombined to form the chromosomes of the child individual. To that end, a one-point crossover mechanism (discussed below) was used.

One point crossover generates a child chromosome by concatenating complement segments of the parent chromosomes. The point of segmentation (called the crossover point) is selected at random. An example crossover is illustrated in Figure 1.

$\mathbf{p}_1$	A_1	B_1	C_1	D_1
$\mathbf{p}_2$	A_2	B_2	C_2	D_2

One possible child chromosome would be (if the crossover point were between the third and fourth gene)

Child	A_1	B_1	C_1	D_2

Figure 1: An Example Crossover

Mutation: Mutation is a process by which a chromosome is altered slightly in order to create a variant of itself. This is done in order to allow the GA to escape local optima by allowing the chromosome (and hence the individual) to move to a different point in the solution space. One possible mutation on a binary bit-string chromosome would flip the value of a random bit, as illustrated in Figure 2.

Before Mutation	0	1	0	0
After Mutation	0	1	0	1

Figure 2: An Example Mutation

2) Evolutionary Strategy: An Evolutionary Strategy (ES) is an evolutionary algorithm similar to a GA. The main difference is that ESs do not use crossover operators. Instead, once the initial random population has been generated, they progress from one generation to the next by selecting the fittest individual of that generation and mutating it several times in random ways in order to form the individuals of the next generation of that population.

II. Previous Work

There are two competing schools of thought in using evolutionary algorithms for dynamic white-box testing:

1) Given all target paths through the CFG of the SUT, run one instance of a GA to learn the inputs for each such path [12]. This will be referred to as OFE (One for Each).
2) Given all target paths through the CFG of the SUT, run one instance of a GA to collectively learn the inputs for all target paths [5, 13, 3].This will be referred to as OFA (One for All).

Note the following definitions of target path and induced path:

1) **Target path**: a path in the CFG of the SUT from the start node to any terminal node that needs to be executed by some vector of inputs in the test suite.
2) **Induced path**: the path in the CFG of the SUT that is observed to have been executed by running the SUT on a vector of input test data.

OFE is inefficient as it encourages redundant learning among two GA instances that have to learn inputs in a small neighborhood of the search space, owing to similar induced paths in the CFG.

On the other hand, OFA is also inefficient. Consider a set of target paths in which most target paths are very similar to each other, but one target path is very dissimilar. In such a scenario, the GA used in OFA would likely converge to the part of the search space containing inputs that induce the similar target paths. It would then have to diverge in the search space and converge again onto a different part of the search space that contains inputs to induce the dissimilar target path. This required diverging can be viewed as "unlearning" or "evolving backwards", which is redundant and therefore inefficient.

III. Methodology

As discussed in section II, using one run of a GA per collection of target paths in order to discover test input data for a SUT is inefficient. At the same time, executing one run of the GA per target path seems inefficient. Therefore, a balance needs to be struck with a hybrid system. Such a hybrid system is presented in this paper.

In the spirit of the work presented in [14] this paper will generate groupings of target paths. However, a run of the GA for each group will not start with a normal random population generation function. Rather, the initial population will be primed in some sense to reflect the context of the grouping of similar target paths. This is somewhat visualized in Figures 3 and 4.

1) The CFG was generated from the SUT by a human agent as an appropriate software library was unavailable for the used implementation.
2) As a preprocessing step to the evolutionary algorithm to follow, the similarity between every pair of target paths extracted from the CFG is computed, using the relative similarity measure shown in eq. (1), derived from the similarity measure described in [14].
3) A random population of test input data is generated with a meaningful chromosomal structure. This chromosomal structure is described in more detail in Section III-B.
4) The target paths are binned into groups based on target paths whose relative similarities are greater than an experimentally determined threshold.
5) The chromosome in the population that has the highest fitness among each group is then considered to be the

seed for the next generation of test input data for that group of target paths.

6) This chromosome is mutated several times in various ways to form the new population for the group of similar target paths.
7) Recursively repeat steps 4 - 6 until the resulting groups of similar target paths are all singletons.
8) Execute a run of the GA per target path (i.e. per singleton group of target paths) until a test input is discovered that induces the target path.
9) Record the discovered test input and the target path and terminate the evolutionary process for that group.

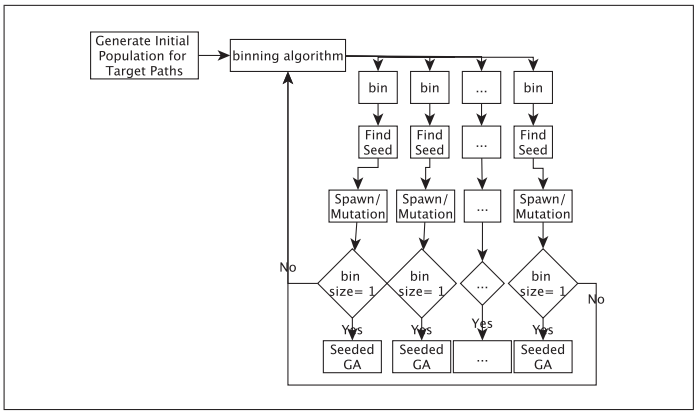

Figure 3: Overview of the Recursive Hybrid Evolutionary Algorithm

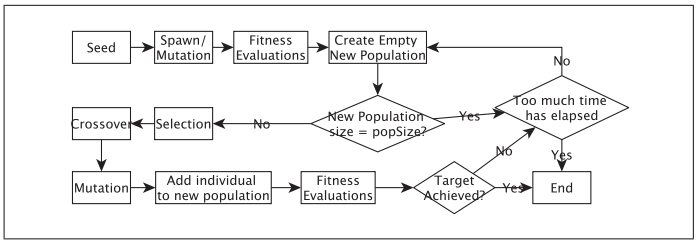

Figure 4: Overview of the Seeded Genetic Algorithm

Despite the drawbacks to executing separate runs of the evolutionary process per group or target paths, the following justifications can be made:

1) This method is highly parallelizable as each run of the evolutionary process per group or target paths in one level of recursion is independent of other similar runs in the same level of recursion.
2) Due to the small size of each group, the evolutionary process to execute a run of the GA would not take as long.
3) Since the initial population for the GA has already undergone several generations evolution, the initial population is already comprised of multiple very fit chromosomes[2]. This would only expedite the process of evolution.
4) Since the GAs themselves can be executed fairly quickly (as explained above), the maximum tardiness of the last machine to finish such a run of the GA in a parallel environment can be reduced.

A. The Benchmark Problems

To benchmark this algorithm against other algorithms, the well known triangle classification program, the bubble-sort program and the min-max algorithm (presented in [3])were used as the SUTs. The python implementation of these programs are presented in Algorithms 1, 2 and 3. All target paths for Algorithm 1 are listed in Table III and all target paths for Algorithm 3 are listed in Table . Algorithm 2 has a total of 64 target paths, which are not listed here due to the sheer number of entries.

Algorithm 1 The Triangle Classification Program (SUT)

```
1  def classify(x,y,z):
2    """
3      Return codes:
4          -1: not a triangle
5        0: equillateral
6        1: isosceles
7        2: scalene
8    """
9
10   if x<y+z and y<z+x and z<x+y:
11     if x!=y and y!=z and z!=x:
12       return 2
13     else:
14       if x==y==z:
15         return 0
16       else:
17         return 1
18   else:
19     return -1
```

The CFGs of these two SUTs are shown in Figures 5 and 6.

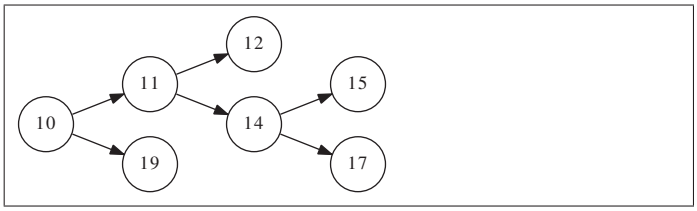

Figure 5: CFG for Triangle Classification Program

These programs have been used as benchmarking SUTs in [3, 7], making them ideal candidates to benchmark the

[2]These chromosomes are at least much more fit than the chromosomes in the initial, randomly generated population

Algorithm 2 The Bubble Sort Program (SUT)

```
1  def bubbleSort(x,y,z):
2    """
3    Store the inputs in a list.
4    Sort the list IN-PLACE by
5      the well-known bubble sort.
6    1. For each element
7    2. For each each adjacent pair
8    3. If the left element
9    is larger than the right
10   4. Swap the two elements
11   5. Return the sorted list
12   """
13
14   L = [x,y,z]
15   for _ in xrange(len(L)):
16     for i in xrange(len(L)-1):
17       if L[i] > L[i+1]:
18         L[i],L[i+1]=L[i+1],L[i]
19   return L
```

Algorithm 3 The Min-Max Program (SUT)

```
1  def minimax(a,b,c):
2    if a>b:
3      if a>c:
4        M = a
5        if b>c:
6          m = c
7        else:
8          m = b
9      else:
10       M = c
11       m = b
12   else:
13     if b>c:
14       M = b
15       if a>c:
16         m = c
17       else:
18         m = a
19     else:
20       M = c
21       m = a
22   return M,m
```

hybrid algorithm presented in this paper. It is also important to note that the proposed algorithm is intended to discover input vectors for large test suites. As a result, its advantages will be better showcased with a SUT with a greater number of target paths. This is one of the reasons why Algorithm 2 was chosen as a benchmark.

B. The Chromosome Structure

Each individual for the evolutionary algorithm presented in this paper encodes a vector of inputs for the benchmarking

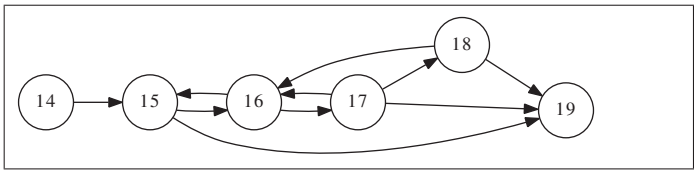

Figure 6: CFG for The Bubble Sort Program

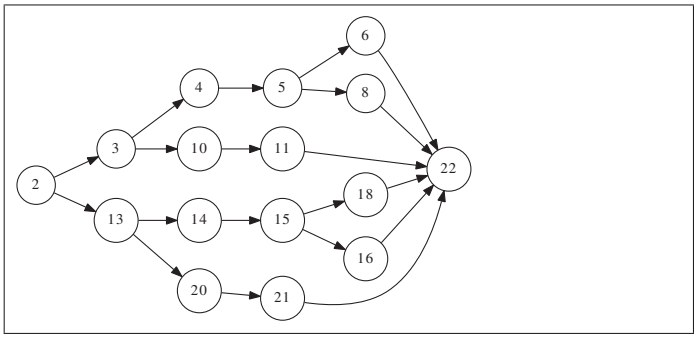

Figure 7: CFG for The Min-Max Program

program used as the SUT. In the style of [7], we restrict the range of the inputs to the inclusive interval $[1, 10]$. The input vector is represented as a 12-bit binary string; a maximum of 4 bits per input parameter is required, in order to be represented as a binary string. Thus, an individual chromosome is comprised of 12 bits, across three 4-bit segments, each of which encodes the value of a particular input. An example of this is shown in Figure 8, by way of a chromosome encoding the input vector <3,4,5> as inputs to the either of the benchmark problems used.

C. Generating the Initial Population

The initial population is generated by creating 1000 unique chromosomes. Each chromosome is generated by three calls to a library function that returns a random integer in the inclusive

Table III: All Target Paths for Algorithm 1

Target Path
(22,31)
(22,23,24)
(22,23,26,27)
(22,23,26,29)

Table IV: All Target Paths for Algorithm 3

Target Path
(2,3,4,7,6,22)
(2,3,4,7,8,22)
(2,3,10,11,22)
(2,13,14,17,16,22)
(2,13,14,17,20,22)
(2,13,20,21,22)

0	0	1	1	0	1	0	0	0	1	0	1
3				4				5			

Figure 8: A Chromosome Encoding the Input Vector <3,4,5>

interval $[1, 10]$. These integers are then converted into 4-bit binary strings (padded with 0s on the left as required) and concatenated to form a single chromosome.

D. Evaluating the Fitness of an Individual

The fitness of an individual is evaluated as follows:

1) The individual's encoding of the three parameters is decoded into three integers
2) The SUT is run on these integers as parameters
3) The lines of code that are executed by the SUT called on these parameters[3] are recorded[4]
4) The induced path is compared to the target path, and a similarity measure is computed[5]
5) The fitness of an individual is directly proportional to how similar it is to the target path (as indicated by the similarity measure)

1) Fitness Evaluation for the ES: For the Evolutionary Strategy side of the hybrid algorithm, the fitness of an individual is computed against a bucket of target paths. This is accomplished as follows:

1) Given an individual i, compute the average relative similarity[6] of the path induced by i against every target path in the bucket
2) The individual with the highest average relative similarity is considered to be the fittest individual for that bucket of target paths

The relative similarity measure is preferred over the non-relative similarity measure, as the relative measure indicates a normalized score. This facilitates comparisons between paths of unequal lengths without skewing the comparison.

2) Fitness Evaluation for the GA: For the Genetic Algorithm side of the hybrid algorithm, the fitness of an individual is computed against a single target path. This is accomplished as follows:

1) Given an individual i, compute the similarity[7] of the path induced by i against the target path
2) The individual with the highest such similarity is considered to be the fittest individual

3) Computing Similarity between Paths: Given two paths p_1 and p_2, the similarity between these paths is computed by the following expression:

$$s(p_i, p_j) = \frac{k-1}{max(|p_i|, |p_j|)} \tag{1}$$

where $1 \leq k \leq max(|p_i|, |p_j|)$ is maximal
and for all $a \leq i \leq k$, the ith node in p_i is exactly the ith node of p_j (for some $1 \leq a \leq max(|p_i|, |p_j|)$). In effect, this measure views both paths as strings, in which each node in the path is considered to be an atomic character. Under that view, k is the length of the largest common substring.

Further, we define a relative similarity

$$s_R(p_i, p_j) = \frac{s(p_i, p_j)}{\sum_{k=0}^{|PATHS|} s(p_i, p_k) + s(p_j, p_k)} \tag{2}$$

For example, if p_i is (22,23,24) and p_j is (22,23,26,27), $s(p_i, p_j) = 0.25$, since the perceived longest common substring is (22, 23), implying $k = 2$.

E. Crossover

A one-point crossover is used, as described in section I-D1. Figure 9 shows the result of such a crossover with the two listed parents and a crossover point of 8 (as indicated by a blank column).

P_1	1	0	0	1	1	0	1	0		1	0	1	1
P_2	0	0	1	1	0	1	0	1		1	0	1	0
C_1	0	0	1	1	0	1	0	1		1	0	1	1
C_2	1	0	0	1	1	0	1	0		1	0	1	0

Figure 9: Example One-Point Crossover

F. Mutation

A mutation is defined for this framework as a single-point mutation (as described in section I-D1). This ensures that one of the bits in the encoding of one of the input parameters is flipped.

G. Algorithmic Parameters

Table V: Parameters for an evolutionary algorithm, run on Algorithm 1

Parameter	Description	Value
popSize	Number of individuals in the population	50
initThreshold	Minimum similarity between two paths to be classified in the same bin for the first round of classification	0.1
threshold	Minimum similarity between two paths to be classified in the same bin for all subsequent rounds of classification	0.7
spawnMutProb	Probability of flipping a single bit in a chromosome when spawning a population from a seed	0.1
crossProb	Probability of crossover in the GA	0.9
mutProb	Probability of mutation in the GA	0.05

[3]This is also known as the path induced by the input parameters

[4]This is done with the use of the trace module provided in the python programming language's standard library

[5]Two different comparisons are used - one for the ES and one for the GA. These are explained in further sections in more detail

[6]shown in equation (2)

[7]shown in equation (1)

Table VI: Parameters for an evolutionary algorithm, run on Algorithm 2

Parameter	Description	Value
popSize	Number of individuals in the population	1000
initThreshold	Minimum similarity between two paths to be classified in the same bin for the first round of classification	0.1
threshold	Minimum similarity between two paths to be classified in the same bin for all subsequent rounds of classification	0.8
spawnMutProb	Probability of flipping a single bit in a chromosome when spawning a population from a seed	0.3
GASpawnMutProb	Probability of flipping a single bit in a chromosome when spawning a population from a seed for the GA	0.15
crossProb	Probability of crossover in the GA	0.9
mutProb	Probability of mutation in the GA	0.05

IV. Results

An algorithm that implemented the second paradigm discussed in Section II was implemented (with the same optimizations as the algorithm presented in this paper) and run, tasked to generate inputs for Algorithms 1 and 2. In addition the recursive hybrid algorithm proposed in this paper was also made to do the same. The total number of fitness evaluations performed by each of these methodologies was recorded along with their relative run times. Ultimately, neither of the algorithms were able to generate all target paths for the bubbleSort program (within the constraints of the given time and computational hardware), but the algorithm proposed in this paper did discover at least as many paths as the other and in a shorter amount of time. The total number of fitness evaluations performed by each algorithm is shown in Table VII, while Table VIII shows the total elapsed time taken by each algorithm to complete execution and Table IX shows the percentage of all target paths induced by the inputs discovered by each algorithm (Tables X, XI and XII present discovered paths).

It is important to note that Table VIII shows a very low run-time for the OFE algorithm to discover all possible paths. This is because it discovers each target path in under one generation as the initial population contains inputs that induce the required target path. Since the hybrid algorithm classifies the target paths into bins before concluding that inputs for certain target paths have been discovered, it is unable to perform such "short-circuiting". This is why the hybrid algorithm requires a much higher run-time in this case.

Further, Table VIII shows that both the OFE algorithm and the Hybrid algorithm require very close amounts of elapsed time in order to complete their discovery of the required target paths. This would appear to negate the superiority of the Hybrid algorithm presented in this paper. However, note that in the reported elapsed time, the OFE algorithm performs 8.7×10^5 fitness evaluations, while the Hybrid algorithm performs 2.91×10^7 evaluations. Thus, while the OFE algorithm performs approximately 300 fitness evaluations per second, the Hybrid algorithm performs 11673 fitness evaluations per second, thereby covering the search space more efficiently.

Table VII: Number of Fitness Evaluations

SUT	OFE	Hybrid
Triangle Classification	300	1900
Bubble Sort	8.7×10^5	2.91×10^7
Min-Max	5.0×10^6	1.5×10^6

Table VIII: Elapsed Run-Time

SUT	OFE	Hybrid
Triangle Classification	1 sec	9 sec
Bubble Sort	2903 sec	2493 sec
Min-Max	296 sec	137 sec

Table IX: Percentage of Target Paths Induced by the Test Suite

SUT	OFE	Hybrid
Triangle Classification	100%	100%
Bubble Sort	9.375%	9.375%
Min-Max	16.67%	66.67%

Table X: Paths discovered by both algorithms for Algorithm 2

Induced Target Paths in Algorithm 2
(14, 15, 16, 17, 18, 16, 17, 18, 16, 15, 16, 17, 18, 16, 17, 16, 15, 16, 17, 16, 17, 16, 15, 19)
(14, 15, 16, 17, 16, 17, 16, 15, 16, 17, 16, 17, 16, 15, 16, 17, 16, 17, 16, 15, 19)
(14, 15, 16, 17, 18, 16, 17, 18, 16, 15, 16, 17, 16, 17, 16, 15, 16, 17, 16, 17, 16, 15, 19)
(14, 15, 16, 17, 18, 16, 17, 16, 15, 16, 17, 16, 17, 16, 15, 16, 17, 16, 17, 16, 15, 19)
(14, 15, 16, 17, 16, 17, 18, 16, 15, 16, 17, 18, 16, 17, 16, 15, 16, 17, 16, 17, 16, 15, 19)

V. Conclusions

The preliminary data presented supports the conclusion that the hybrid algorithm presented in this paper outperforms an existing method in run-time. This is because it harnesses a logarithmic decay in the computational cost of the fitness function owing to the recursive classification of target paths into sub-bins.

The data also supports that conclusion that this method is better suited for testing SUTs with many paths, each of which have many constraints on them.

Further, since the runs of the ESs and GAs at any given level of bin classification are independent of each other, this method is highly parallelizable. In addition, the binning allows for an approximately logarithmic decay in the number of target paths to be included in the fitness function. This implies that the fitness values of individuals in a population is computed faster as elapsed execution time progresses. This, however, is not the

Table XI: Paths discovered by the Hybrid Algorithm but not OFE for Algorithm 2

Induced Target Paths in Algorithm 2
(14, 15, 16, 17, 16, 17, 18, 16, 15, 16, 17, 16, 17, 16, 15, 16, 17, 16, 17, 16, 15, 19)

Table XII: Paths discovered by the OFE but not the Hybrid Algorithm for Algorithm 2

Induced Target Paths in Algorithm 2
(14, 15, 16, 17, 16, 17, 16, 15, 16, 17, 18, 16, 17, 18, 16, 15, 16, 17, 18, 16, 17, 16, 15, 19)

case for the other method, which maintains a constant-sized set of target paths throughout execution. This is one reason why the hybrid algorithm completed execution significantly faster, despite having computed significantly more fitness values.

The bubble sort algorithm was used as a benchmarking SUT to illustrate this. With 64 target paths, the existing method was required to compare the path induced by each individual in every generation of the population with 64 target paths. A population size of 1000 therefore drives 64000 path comparisons per generation. However, with the hybrid, a GA is only invoked on a single target path. Thus, even with 1000 individuals in the population, only 1000 fitness evaluations are made. Further, due to the threshold values, the logarithmic nature of the decay of bin sizes forces the hybrid algorithm to perform progressively fewer fitness evaluations on every successive call to the ES on the target paths in a bin. Ultimately, the algorithm presented in this paper discovers inputs that induce a set of paths that contains at least all the paths induced by the inputs discovered by the other method. These induced paths are shown in Table X. It is of interest to note that fewer fitness evaluations are performed by the hybrid algorithm on the Min-Max SUT. This is not anomalous. Rather, it is an artifact of the algorithm refusing to perform fitness evaluations after an individual that induces the required target path has been discovered. Thus, since the path coverage is much higher in the case of the hybrid algorithm, there are more occassions when it stops early, explaining the lower number of fitness evaluations.

VI. Future Work

Further advancements of this work could include the development of a similarity measure that does not require experimental values for the threshold used to classify target paths into bins. Such a measure would be on a standardized scale that may require some prior knowledge about the target paths, so that a threshold may be set without much experimentation.

References

[1] B. Korel, "Automated Software Test Data Generation," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 16, no. 8, pp. 870–879, 1990. [Online]. Available: http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7302/ReadingMaterial/Korel90.pdf

[2] E. Dustin, J. Rashka, and J. Paul, *Automated software testing: introduction, management, and performance*. Addison-Wesley Professional, 1999.

[3] M. a. Ahmed and I. Hermadi, "GA-based multiple paths test data generator," *Computers & Operations Research*, vol. 35, no. 10, pp. 3107–3124, Oct. 2008. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S0305054807000251

[4] G. J. Myers, "The art of software testing, Publication info."

[5] M. Pei, E. D. Goodman, Z. Gao, and K. Zhong, "Automated software test data generation using a genetic algorithm," *Michigan State University, Tech. Rep*, no. 1, pp. 1–15, 1994.

[6] C. S. Pasareanu, N. Rungta, and W. Visser, "Symbolic execution with mixed concrete-symbolic solving," *Proceedings of the 2011 International Symposium on Software Testing and Analysis - ISSTA '11*, p. 34, 2011. [Online]. Available: http://portal.acm.org/citation.cfm?doid=2001420.2001425

[7] D. Berndt, J. Fisher, L. Johnson, J. Pinglikar, A. Watkins, and I. P. Management, "Breeding software test cases with genetic algorithms," in *System Sciences, 2003. Proceedings of the 36th Annual Hawaii International Conference on*. IEEE, 2003, pp. 10—-pp.

[8] C.-f. Juang, "A Hybrid of Genetic Algorithm and Particle Swarm Optimization for Recurrent Network Design," vol. 34, no. 2, pp. 997–1006, 2004.

[9] J. Hunt, "Testing control software using a genetic algorithm," *Engineering Applications of Artificial Intelligence*, vol. 8, no. 6, pp. 671–680, 1995.

[10] D. E. Goldberg and K. Deb, "A comparative analysis of selection schemes used in genetic algorithms," *Urbana*, vol. 51, pp. 61 801–62 996, 1991. [Online]. Available: http://pdf.aminer.org/000/212/484/a_comparative_analysis_of_selection_schemes_used_in_genetic_algorithms.pdf

[11] X. Yao, *Evolutionary computation: theory and applications*. World Scientific Publishing Company Incorporated, 1999.

[12] B. Jones, H.-H. Sthamer, and D. Eyres, "Automatic structural testing using genetic algorithms," *Software Engineering Journal*, vol. 11, no. 5, p. 299, 1996. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/sej.1996.0040

[13] D. Gong, T. Tian, and X. Yao, "Grouping target paths for evolutionary generation of test data in parallel," *Journal of Systems and Software*, 2012.

[14] D. Gong, W. Zhang, and X. Yao, "Evolutionary generation of test data for many paths coverage based on grouping," *Journal of Systems and Software*, vol. 84, no. 12, pp. 2222–2233, Dec. 2011. [Online]. Available: http://linkinghub.elsevier.com/retrieve/pii/S016412121100152X