Paper Title\* (use style: *paper title*)

Subtitle as needed (*paper subtitle*)

Authors Name/s per 1st Affiliation *(Author)*

line 1 (of *Affiliation*): dept. name of organization

line 2: name of organization, acronyms acceptable

line 3: City, Country

line 4: e-mail address if desired

Authors Name/s per 2nd Affiliation *(Author)*

line 1 (of *Affiliation*): dept. name of organization

line 2: name of organization, acronyms acceptable

line 3: City, Country

line 4: e-mail address if desired

*Abstract*—Automatic path coverage test data generation is an undecidable problem and genetic algorithm (GA) has been used to test data generation since 2000. This paper presents a method for optimizing GA efficiency by identifying the most critical path clusters in a test function. We do this by using the static program analysis to find all the paths having the path conditions with low probability in generating coverage data, then basing on these path conditions to adjust the procedure of generating new populations in GA. The proposed approach is also applied some test functions. Experimental results show that improved GA which can generate suitable test data have higher path coverage than the traditional GA.

Keywords—generic algorithm; path coverage testing; automatic test data generation

# Introduction

Software quality becomes more important than ever and software testing is the most significant measure for it. However, software testing is very laborious and costly due to the fact that it is mostly made by manual [1]. In general, software testing accounts for approximately 50 percent of the elapsed time and more than 50 percent of the total cost in software development [2]. Thus, automated software testing is a promising way to cut down time and cost.

Automatic structural test data generation is a crucial problem in software testing automation and its implementation can not only significantly improve the effectiveness and efficiency but also reduce the high cost of software testing. We focus on path coverage test data generation in respect that various structural test data generation problem can be transformed into a path coverage test data generation problem. Furthermore, path coverage testing strategy can detect almost 65 percent of errors in program under test [3].

Although path coverage test data generation is an undecidable problem [4], researchers still attempt to develop various methods and have made some progress. These methods can be classified into two types: static methods and dynamic methods.

Static methods include symbolic execution [5] and domain reduction [6, 7] etc. These methods suffer from a number of problems when it handles indefinite loops, array, procedure calls and pointer references [8].

Dynamic methods include random testing, local search approach [9], goal-oriented approach [10], chaining approach [11] and evolutionary approach [8, 12-14]. Since values of input variables are determined when programs execute, dynamic test data generation can avoid those problems with that static methods are confronted.

As a robust search method in complex spaces, genetic algorithm (GA) was applied to test data generation in 1992 [12] and evolutionary approach has been a burgeoning interest since then. Related works [8, 15, 16] indicate that GA-based test data generation outperforms other dynamic approaches e.g. random testing and local search.

As far as we know, even though GA-based test data generation already proved its efficiency in generating test data for dynamic approaches, it still has to face difficulties when applying to path coverage, which are for the test function having test path with low probability in generating coverable test data, traditional GA can not generate test data which can cover these difficult paths.

This paper gives the proposal to improve traditional GA in generating test data which can cover all the paths in a test function. It combines static program analysis with GA. The static program analysis step is applied to find out paths of the test function which are difficult to be covered. After that, basing on the path condition of these difficult paths, adjusts the procedure of generating new populations in GA.

This paper is organized as follows: Section 2 presents some theoretical background to understanding this research. Section 3 summarizes some related works, and Section 4 presents the proposed approach in detail. Section 5 gives a case study to illustrate our proposed approach. Section 6 shows the experimental results and discussion. Section 7 concludes the paper.

# BACKGROUND

## Path testing

The objective of path testing is to search for a collection of test cases (inputs to a program) that between them lead to the traversal of all logical paths through the program. In general, path testing process consists of two major steps: target paths generation and test data generation. Both of them will be mentioned in our proposal of this paper.

#### Target paths generation: Target paths generation means identifying a set of logical execution pathways through the program, that we hope should all be exercised during testing. The source code is needed to construct its logical control flow, which can be presented in a control flow graph (CFG). This graph can be automatically generated by using appropriate programming language grammar in which the program is written. From the CFG, the different logical paths through the program need to be enumerated. A logical path is a particular flow of execution through the program, which is determined by the decisions made at each decision point between the program’s entry point and its exit point.

#### Test data generation: Generating test data that fulfill path coverage is the main task in path testing. It is the process of creating test data, either heuristically or randomly. In a heuristic approach, the process is guided by some rules to search for required test data; the alternative is that random test data is generated.

## Generic algorithm

The basic concepts of genetic algorithm (GA) were developed by Holland [17]. GA are commonly applied to a variety of problems involving search and optimisation. GA search methods are rooted in the mechanisms of evolution and natural genetics. GA draw inspiration from the natural search and selection processes leading to the survival of the fittest individuals. GA generate a sequence of populations by using a selection mechanism, and use crossover and mutation as search mechanisms.

The principle behind GA is that they create and maintain a population of individuals represented by chromosomes (essentially a character string analogous to the chromosomes appearing in DNA). These chromosomes are typically encoded solutions to a problem. The chromosomes then undergo a process of evolution according to rules of selection, crossover and mutation.

Each individual in the environment (represented by a chromosome) receives a measure of its fitness in the environment. Reproduction selects individuals with high fitness values in the population, and through crossover and mutation of such individuals, a new population is derived in which individuals may be even better fitted to their environment. The process of crossover involves two chromosomes swapping chunks of data (genetic information) and is analogous to the process of sexual reproduction. Mutation introduces slight changes into a small proportion of the population and is representative of an evolutionary step. The structure of a simple GA is given below.

Simple Genetic Algorithm (){

initialize population;

evaluate population;

while (stopping criteria not reached) {

select solutions for next population;

perform crossover and mutation;

evaluate population;

}

}

The algorithm will iterate until the population has evolved to form a solution to the problem, or until a maximum number of iterations have taken place (suggesting that a solution is not going to be found given the resources available).

# RELATED WORK

The path coverage literature using GA started with Lin and Yeh [18] in 2000. They extended Jones et al.'s work [19] from branch coverage to path coverage. The ordinary (weighted) Hamming distance was extended to handle different ordering of target paths that have the same branches. The fitness function is called SIMILARITY, which computes similar items with respect to their ordering within two different paths between actual executed path and the target path. Only one program was used to test the approach, i.e. simple triangle classifier. They reported that the approach outperformed random search. However, in this method, evaluation function must be called many many times in order to generate the test data for the most difficult path to be covered.

Bueno et al. [20] proposed an approach that utilizes control and data flow dynamic information to achieve path coverage testing using GA. In addition, the work also tackled the detection of infeasible paths by monitoring the progress of evolutionary search. The fitness function was formulated by number of coincidence branches and the normalized branch predicate value at which the actual executed path starts to deviate from the target path. Six small test programs were used to validate the approach, with 10 repetitions each to minimize random variations. Two execution modes were used, i.e. one with initialized population and the other with a random initial population. The experiment results were promising.

In 2003, Hermadi and Ahmed [21] presented evolutionary test data generation for path testing using multiple paths. Prior to this work, almost all of the evolutionary test data generators only sought to cover a single target path at a time. The fitness function used the number of matching branches and branch predicate values using Korel's fitness function [9]. It also considered path traversal techniques, neighbourhood influence, weighting, and normalization. Three small programs were used to validate the approach: minimum-maximum finder, triangle classifier, and a combination of both of them. Results were more effective and efficient by tackling multiple paths at a time.

In 2008, Ahmed and Hermadi [23] extended Hermadi and Ahmed's work of 2003 [21]. The extensions were adding a rewarding scheme and using a more efficient test data generator. A total of 32 fitness function variations were tested empirically and analysed to determine which was the best. There were 7 test programs used in the experiments. The results demonstrated that the approach was better compared to other existing work.

In the same year, Chen and Zhong [24] developed a multi-population genetic algorithm for path testing. This work has been improving GA-based path testing as described in Section 2.2. The work reported that the proposed approach outperformed a simple genetic algorithm based approach, using the triangle classifier as the test function. Similar to our approach, this paper also targets finding the test data to cover path conditions of the most difficult path to be covered in test function. As it approached the parallel processing, test data generating time is better than traditional GA, however the number of test data generation is still high (requires 21,073 test data by average).

In [25], Srivastava P.R and Kim T have presented a method for optimizing software testing efficiency by identifying the most critical path clusters in a program. The software under test is converted into a CFG. Weights are assigned to the edges of the CFG by applying 80-20 rule. 80 percentage of weight of incoming credit is given to loops and branches and the remaining 20 percentage of incoming credit is given to the edges in sequential path. The summation of weights along the edges comprising a path determines criticality of path. Higher the summation more critical is path and therefore must be tested before other paths. In this way by identifying most critical paths that must be tested first, testing efficiency is increased.

# PROPOSED APPROACH

In order to generate test data which can cover the paths having the lowest coverable probability, we propose 2 steps approach as in the above flow chart:



**Fig. 1.** Flow chart of our proposed approach

Step 1: Peform static program analysis to create the target paths collection from CFG of input test function. Then from the target paths determine the path conditions of difficult paths.

Step 2: Execute GA to automatically generate test cases as below.

#### Representation: Depend on the type of input parameters of test function, GA uses a real or integer vector as a chromosome x = (x1, x2,…, xn) to represent values of the input variables. The length of the vector depends on the required precision and the domain length for each input variable.

#### Initial population: At first, it needs to identify a fixed popsize number is the number of chromosome in a population. After that is the initialization of a randome value for all chromosomes in this initial population.

#### Evaluation function: The algorithm evaluates each test case by executing the program with it as input, and recording the executed path in the program that are covered by this test case. The fitness value eval(xi) for each chromosome xi (i = 1,…, xn) is calculated as follows:

if (executed path target paths){

eval(xi) = 0;

remove this path from the target paths;

}else{ eval(xi)=

}

The distance between 2 different paths of given test function is defined as below:

distance(pathi,pathj)=

#### Selection: A selection scheme is applied to determine how individuals are chosen for mating based on their fitness. Fitness can be defined as a capability of an individual to survive and reproduce in an environment. Selection generates the new population from the old one, thus starting a new generation. Each chromosome is evaluated in present generation to determine its fitness value. This fitness value is used to select the better chromosomes from the population for the next generation.

#### Crossover: After selection, the crossover operation is applied to the selected chromosomes. It involves swapping of values of vector x = (x1, x2,…, xn) between two chromosomes. This process is repeated with different parent chromosomes until the next generation has enough chromosomes. After crossover, the mutation operator is applied to a randomly selected subset of the population.

#### Mutation: Mutation alters chromosomes in small ways to introduce new good traits. It is applied to bring diversity in the population.

#### Adjustment: After executing the mutation, in order to generate test cases which can cover the most difficult paths, we need to adjust the values of each chromosome in the population. The adjustment will be executed as follows:

• Browse the list of path conditions of the difficult paths to be covered.

• If any value of the chromosome can nearly satisfy this path condition, adjust it so that it can satisfy such condition.

# CASE STUDY

Consider the test function Tritype tA2008 (simple triangle classifier) [18, 22] which determines whether three given numbers that represent three lengths on a plane form a scalene, isosceles, equilateral, or not a triangle.

int Tritype(int a, int b, int c){

int trityp = 0;

if ((a + b > c) && (b + c > a) && (c + a > b)){

if ((a != b) && (b != c) && (c != a))

trityp = 1; // Scalene

else

if (((a == b) && (b != c)) || ((b == c) &&

(c != a)) || ((c == a) && (a != b)))

trityp = 2; // Isosceles

else

trityp = 3; // Equilateral

} else

trityp = -1; // Not a triangle

return trityp;

}



**Fig. 2.** The CFG of above Tritype function

This CFG consists of 4 paths as below:

{[1,F]} // path 1 (Not Triangle)

{[1,T], [2,F], [3,F]} // path 2 (Equilateral)

{[1,T], [2,F], [3,T]} // path 3 (Isosceles)

{[1,T], [2,T]} // path 4 (Scalence)

Assuming that the input parameters a, b, c take the positive integer values in the range of a, b, c ∈ [1, 1000]. Each parameter can take 1000 values, so the space of input parameter will be 10003. There are 1000 triagles having equal sides which are (1, 1, 1), (2, 2, 2)…(1000, 1000, 1000). Therefore, the probability that one test data can cover path 2 (Equilateral) is 1000/10003 = 0.000001. Also through a similar calculation, we will have a table showing the generation probability of path coverage data as follows:

1. Table Type Styles

| **Classification** | **Probability** |
| --- | --- |
| Path 4 (Scalence) | 0.4985015 |
| Path 3 (Isosceles) | 0.001998 |
| Path 2 (Equilateral) | 0.000001 |
| Path 1 (Not Triangle) | 0.4994995 |

From the experiments, we found out that if only using the traditional GA to generate data for Tritype function, we can not generate the test data for path 2 (Equilateral), which is the most difficult path to be covered.

GA started....

Path 4: a = 737 b = 241 c = 615

Objective call: 1

Path 1: a = 389 b = 69 c = 133

Objective call: 3

Path 3: a = 764 b = 764 c = 819

Objective call: 81

Objective call: 10000

GA completed./.

Now we apply our proposal for this test function. At first determine the path conditions of difficult paths to be covered.

1. Table Type Styles

| **Classification** | **Probability** | **Path condition** |
| --- | --- | --- |
| Path 4 (Scalence) | 0.4985015 |  |
| Path 3 (Isosceles) | 0.001998 | a==b || b==c || c==a |
| Path 2 (Equilateral) | 0.000001 | a == b == c |
| Path 1 (Not Triangle) | 0.4994995 |  |

Then perform GA to automatically generate test cases. With the above Tritype function, as input parameter was a set of three numbers (a, b, c) each generated test case will be perofrmed by a chromosome, which is a vector x = (x1, x2, x3). The initial target paths list also includes 4 paths of the test function.

Now we have to calculate the distance between each path in given test function. Using above proposed formula, for example, we have some distance as below:

distance(path 1, path 2) = 3

distance(path 2, path 3) = 0.3333

At last we adjust each value of the chromosome for the above identified path condition of difficult path. Adjustment is executed as follows:

for i = 1 to popsize do {

if ((abs(chromosome[i].x1 - chromosome[i].x2) <= epsilon) &&

(abs(chromosome[i].x1 - chromosome[i].x3) <= epsilon)) {

// Adjustment for path 2 (Equilateral)

chromosome[i].x2 = chromosome[i].x1;

chromosome[i].x3 = chromosome[i].x1;

}

}

With these adjustments, the returned results after executing GA are as follows:

GA started....

Path 4: a = 737 b = 241 c = 615

Objective call: 1

Path 1: a = 389 b = 69 c = 133

Objective call: 3

Path 3: a = 764 b = 764 c = 819

Objective call: 81

Path 2: a = 900 b = 900 c = 900

Objective call: 3198

GA completed./.

## Authors and Affiliations

The template is designed so that author affiliations are not repeated each time for multiple authors of the same affiliation. Please keep your affiliations as succinct as possible (for example, do not differentiate among departments of the same organization). This template was designed for two affiliations.

### For author/s of only one affiliation (Heading 3): To change the default, adjust the template as follows.

#### Selection (Heading 4): Highlight all author and affiliation lines.

#### Change number of columns: Select the Columns icon from the MS Word Standard toolbar and then select “1 Column” from the selection palette.

#### Deletion: Delete the author and affiliation lines for the second affiliation.

### For author/s of more than two affiliations: To change the default, adjust the template as follows.

#### Selection: Highlight all author and affiliation lines.

#### Change number of columns: Select the “Columns” icon from the MS Word Standard toolbar and then select “1 Column” from the selection palette.

#### Highlight author and affiliation lines of affiliation 1 and copy this selection.

#### Formatting: Insert one hard return immediately after the last character of the last affiliation line. Then paste down the copy of affiliation 1. Repeat as necessary for each additional affiliation.

#### Reassign number of columns: Place your cursor to the right of the last character of the last affiliation line of an even numbered affiliation (e.g., if there are five affiliations, place your cursor at end of fourth affiliation). Drag the cursor up to highlight all of the above author and affiliation lines. Go to Column icon and select “2 Columns”. If you have an odd number of affiliations, the final affiliation will be centered on the page; all previous will be in two columns.

## Identify the Headings

Headings, or heads, are organizational devices that guide the reader through your paper. There are two types: component heads and text heads.

Component heads identify the different components of your paper and are not topically subordinate to each other. Examples include Acknowledgments and References and, for these, the correct style to use is “Heading 5”. Use “figure caption” for your Figure captions, and “table head” for your table title. Run-in heads, such as “Abstract”, will require you to apply a style (in this case, italic) in addition to the style provided by the drop down menu to differentiate the head from the text.

Text heads organize the topics on a relational, hierarchical basis. For example, the paper title is the primary text head because all subsequent material relates and elaborates on this one topic. If there are two or more sub-topics, the next level head (uppercase Roman numerals) should be used and, conversely, if there are not at least two sub-topics, then no subheads should be introduced. Styles named “Heading 1”, “Heading 2”, “Heading 3”, and “Heading 4” are prescribed.

## Figures and Tables

#### Positioning Figures and Tables: Place figures and tables at the top and bottom of columns. Avoid placing them in the middle of columns. Large figures and tables may span across both columns. Figure captions should be below the figures; table heads should appear above the tables. Insert figures and tables after they are cited in the text. Use the abbreviation “Fig. 1”, even at the beginning of a sentence.

1. Table Type Styles

| Table Head | Table Column Head | | |
| --- | --- | --- | --- |
| Table column subhead | Subhead | Subhead |
| copy | More table copya |  |  |

1. Sample of a Table footnote. (*Table footnote*)
2. Example of a figure caption. (*figure caption*)

Figure Labels: Use 8 point Times New Roman for Figure labels. Use words rather than symbols or abbreviations when writing Figure axis labels to avoid confusing the reader. As an example, write the quantity “Magnetization”, or “Magnetization, M”, not just “M”. If including units in the label, present them within parentheses. Do not label axes only with units. In the example, write “Magnetization (A/m)” or “Magnetization {A[m(1)]}”, not just “A/m”. Do not label axes with a ratio of quantities and units. For example, write “Temperature (K)”, not “Temperature/K”.

##### Acknowledgment *(Heading 5)*

The preferred spelling of the word “acknowledgment” in America is without an “e” after the “g”. Avoid the stilted expression “one of us (R. B. G.) thanks ...”. Instead, try “R. B. G. thanks...”. Put sponsor acknowledgments in the unnumbered footnote on the first page.

##### References

The template will number citations consecutively within brackets [1]. The sentence punctuation follows the bracket [2]. Refer simply to the reference number, as in [3]—do not use “Ref. [3]” or “reference [3]” except at the beginning of a sentence: “Reference [3] was the first ...”

Number footnotes separately in superscripts. Place the actual footnote at the bottom of the column in which it was cited. Do not put footnotes in the reference list. Use letters for table footnotes.

Unless there are six authors or more give all authors’ names; do not use “et al.”. Papers that have not been published, even if they have been submitted for publication, should be cited as “unpublished” [4]. Papers that have been accepted for publication should be cited as “in press” [5]. Capitalize only the first word in a paper title, except for proper nouns and element symbols.

For papers published in translation journals, please give the English citation first, followed by the original foreign-language citation [6].

1. G. Eason, B. Noble, and I. N. Sneddon, “On certain integrals of Lipschitz-Hankel type involving products of Bessel functions,” Phil. Trans. Roy. Soc. London, vol. A247, pp. 529–551, April 1955. *(references)*
2. J. Clerk Maxwell, A Treatise on Electricity and Magnetism, 3rd ed., vol. 2. Oxford: Clarendon, 1892, pp.68–73.
3. I. S. Jacobs and C. P. Bean, “Fine particles, thin films and exchange anisotropy,” in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.
4. K. Elissa, “Title of paper if known,” unpublished.
5. R. Nicole, “Title of paper with only first word capitalized,” J. Name Stand. Abbrev., in press.
6. Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, “Electron spectroscopy studies on magneto-optical media and plastic substrate interface,” IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].
7. M. Young, The Technical Writer’s Handbook. Mill Valley, CA: University Science, 1989.