Automatic Search-Based Software Test Data Generation via Dynamic Execution with Heuristic Optimization Algorithm Classifier

by

Danyang Zhao

A research paper
presented to the University of Waterloo
in fulfillment of the
research paper requirement for the degree of
Master of Mathematics
in
Computational Mathematics

Waterloo, Ontario, Canada, 2025

Abstract

Generating test data that cover all newly developed code has been an important step in the software development process. Existing research has explored ways to automatically generate test data to reduce costs and time in this process. This research paper proposes a classifier that aims to find a more efficient automatic test data generation method through dynamic executions of Python-written programs. The automatic generation process converts conditional statements to function minimization problems that can be solved by optimization algorithms, and then dynamic execution updates the local variable values. This classifier assigns the best optimization algorithm to each conditional statement based on the patterns observed during the experiments. During the generation phase, a heuristic approach isolates the variables affecting the current conditional statement to further improve efficiency. The implementation of this classifier and the results of coverage by iterations and coverage by runtime are shown in this research paper. We also benchmark the results against the random test data generation and the Least Squares method, demonstrating the necessity of using optimization algorithms for effective test data generation. The results of the experiments validate that the integration of the classifier improves conditional-decision coverage while reducing runtime.

Acknowledgements

I would like to thank my supervisor, Professor Saeed Ghadimi, for his invaluable guidance and continuous support throughout my research.

I would also like to thank Professor Giang Tran for her feedback and for taking the time to review my research paper.

Finally, I am grateful to my family and friends for their unwavering support during this journey.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

As technological applications continue to develop and improve people's productivity, there has been increasing demand to assess their adequacy. Detecting faults in a newly developed program before its release is an important step to improve the user experience and the company's reputation. However, manual program testing has been costly and inefficient. It requires manual test case setup and may not cover all cases. For newly implemented features, it is important to set up test cases that cover every conditional statement. This step may become redundant and tedious for testers. Therefore, an automatic test case generator can assist in testing and provide a comprehensive preliminary test set in addition to the testers' numerical accuracy tests.

A test adequacy criteria is needed to evaluate adequacy [22]. A problem is designed to generate test cases based on the selected adequacy criteria. Since not every problem is solvable, heuristics are generally used to generate test cases to find the method that provides a better result but does not guarantee success in 100%.

The current literature shows that there are generally two ways of evaluating a program: symbolic or dynamic execution. Symbolic evaluation often involves analyzing all possible paths in a program, then generating test cases that execute the selected path without accessing the numeric values of the variables [14]. The dynamic execution approach [22] does not symbolically evaluate the whole program upfront, but executes the program directly, and only considers the current step in the program, not the entire path. This research will focus on the dynamic execution approach.

There has been significant interest in reducing testing costs and finding the best method to automatically generate test cases. Search-based approaches have been used extensively, for example, gradient descent method can be used for continuous and smooth conditions, particle swarm algorithm [28](heuristic search approach) and genetic algorithm [22](stochastic search approach) can be used for discrete conditions, such as boolean conditions that contain non-numeric variables (strings, functions, etc.). However, current literature shows that gradient descent cannot be applied to discrete problems, particle swarm has the issue of stopping at a local minimum, while genetic algorithm needs crossover and mutation and eliminates part of the population in the process.

We are proposing a hybrid method that classifies the type of the condition, then applies different optimization algorithms to generate a test case that can satisfy the condition. This new methodology aims to increase the speed of test case generation and increase the conditional coverage rate.

Condition coverage testing is used to test the effectiveness of the proposed method. For each "if" or "else if" statement, both of its true and false branches count as conditions that need to be met. The coverage is calculated as the number of conditions met / total number of conditions.

1.2 Problem statement

This research focuses on automatic test data generation using the dynamic execution method. The dynamic execution method reduces the generation process to a function minimization problem. Therefore, the problem is to minimize the objective functions of all conditional statements. While the program is being executed, the minimization problem is performed for the conditional statement using the current the values of the local variables. Each conditional statement is converted to an objective function:

$$\min_{X} f(X), X \in \{x_1, x_2, x_3, ...\}, \tag{1.1}$$

where X is a set of input parameters that are involved in the current conditional statement, and f(X) is the objective function that the conditional statement is converted to. The variables x_1, x_2, x_3 stand for input parameters that form the test data.

For each conditional statement, if it is a conjunction of sub-conditions, e.g. it contains a logical AND, we will minimize the sum of objective values of each sub-condition; while if it is a disjunction of sub-conditions, e.g., it contains a logical OR, we will only minimize the objective value of the first operand of the OR expression. The mathematical formulas are as follows:

Logical AND:

$$\min_X \sum_i f_i(X_i), X \in \{X_1, X_2, ..., X_i\}, X_i \in \{x_{i1}, x_{i2}, x_{i3}, ...\}, \tag{1.2}$$

where the index i represents a sub-condition in the current conditional statement, $f_i(X_i)$ is the objective function for the corresponding sub-condition. X_1, X_2, X_3 represent the set of input parameters of the sub-condition, and x_{i1}, x_{i2}, x_{i3} represent each input parameter in the corresponding set i.

For example, the conditional statement if ((i + j <= k) and (j + k <= i)) will be converted to $\min_{i,j,k}\{(i + j - k) + (j + k - i)\}$.

Logical OR:

$$\min_X f_1(X_1), X \in \{X_1, X_2, ..., X_i\}, X_1 \in \{x_{11}, x_{12}, x_{13}, ...\}, \tag{1.3}$$

where $f_1(X1)$ is the objective function for the first operand of the OR expression, X_1 is the set of parameters of this sub-condition, and x_{11}, x_{12}, x_{13} represent input parameters in this set.

For example, the conditional statement if ((i <= 2) or (j <= 0) or (k <= 0)) will be converted to $\min_i i - 2$, if i > 2.

The constraints of this system are the pre-defined range for the input parameters of the program and the parent conditional statements.

1.3 Related works and research gaps

This research focuses on path-wise test data generators that take inputs and test criteria to produce the desired test data [14]. Two methods are popular for path-wise test data generation: symbolic evaluation [2][9] and dynamic execution [14][22].

1.3.1 Symbolic Evaluation

Symbolic evaluation allows variable values to be non-numeric, for example, elementary symbolic values, arithmetic operators, and can also be a combination with numeric values

[9]. Some early research shows the direction of the evolution of this method. Clarke (1976) [3] describes a system that automatically generates test cases through a symbolic evaluation of each path. The symbolic evaluation can detect path infeasibility efficiently. The DISSECT system [9] requires both the original program and the user defined commands. It can draw a conclusion of the types of errors detected, but it might also require more user input to increase its ability to detect errors. The SELECT system [2] can generate test data or determine the correctness of a path but requires users' inputs in some circumstances, which is not yet automated.

1.3.2 Dynamic Execution

One of the earliest dynamic execution methods was investigated in [14], which identifies the feasible paths of a program and then uses function minimization methods to output the results that can traverse each path. The results show that this method improves effectiveness by overcoming the limitation of symbolic evaluation with variables that are unknown in advance and only become available during the execution of the program. This is a significant improvement as these dynamically changing variables are frequently used in programs. The function minimization method used is the direct-search method, which is similar to applying gradient descent on each input variables individually.

This paper points out that the current function minimization method has some limitations as it might end with a local minimum, not a global minimum. Following this paper, more research was done to explore other function minimization (optimization) methods that address this limitation.

Genetic Algorithm

Michael et al. [23][22] first discussed the use of genetic algorithms (GAs) for the automatic generation of software test data, following [14]. They show that GAs perform better with higher coverage and fewer iterations because they do not get trapped at local minimums like gradient descent algorithms (GDs). They also propose that, while searching for results of one requirement, the intermediate results may also happen to satisfy other requirements, which could save time.

Particle Swarm Optimization

The particle swarm optimization method (PSO) was first introduced in 1995 [13]. Windisch et al. (2007) [28] first applied particle swarm optimization to software testing. They state that PSO is simpler and easier to fine-tune than GA.

Since then, the particle swarm optimization technique has been widely used in automatic test data generation with different testing criteria. The PSODGT (Particle Swarm Optimization Data Generation Tool) [12] first adopts the condition-decision coverage, while [16] used the path coverage. In [12], they state that the PSO is better at searching locally and has a higher convergence speed compared to the GA, which further proves the significance of applying PSO in the test data generation field.

More research was also done to modify and improve the application of particle swarm optimization in the field of automatic test data generation. Wang et al. (2023) [27] discussed the use of a combination of particle swarm optimization (PSO) and GA for the generation of test data. They prove that this new algorithm performs better as it has a more diversified population than the genetic algorithm technique. However, their experiments were based on the efficiency of their proposed algorithm applied on the most complex test path from 3 chosen small programs, not the entire program.

1.3.3 Testing Adequacy Criteria

The generation of test cases usually requires testing adequacy criteria to determine the stop point. For the DISSECT system [9], it identifies the exact types of errors. For the GADGET tool in [22] and the PSODGT tool in [12], they use the decision-conditional testing criteria. In [28], they use the branch coverage.

One of the important measures of testing effectiveness is the code coverage. The code coverage indicates whether all test cases can successfully execute the whole program, which also implies the reliability of the test set [20]. The common testing criteria that has been used are decision-conditional testing and branch testing. The decision-conditional testing is important for software testing because it is usually required for testers to test all newly added code. For the decision-conditional criteria, the generated test data ensures that all code has been tested. Therefore, when the automatic test data generation satisfies the decision-conditional testing criteria, it sets up a complete preliminary dataset to assist with testers' special test data.

1.3.4 Summary

Both the symbolic evaluation and the dynamic execution methods can effectively perform automatic test case generation. However, the symbolic evaluation can detect infeasible paths more efficiently, while the dynamic execution needs to perform function minimization until the stopping criteria has been met. In addition, only the dynamic execution has access to the actual variable values while the program is executing.

Most existing works have tested their algorithms on simple programs with only numerical variables.

In this paper, an automatic dynamic test data generation algorithm is introduced to utilize all of GD, GA and PSO where they are most efficient and can handle more types of input parameters.

This search-based test case generation does not guarantee the correctness of the outputs of the program, but aims to find test cases that reach all conditional statements thus avoiding the existence of unhandled exceptions.

1.4 Contributions

This research extends the automatic test data generator to programs written in the Python language. For Python programs, an existing difficulty is that the types of the input parameters are not required to be pre-defined. As a result, before the values of the input parameters are actually defined, their types are unknown. To overcome this difficulty, we ask the user to pre-define the parameter types in advance, so that the automatic test data generator can generate test data of the correct type.

As mentioned in [22], applying the automatic test data generator on larger and more complex programs is important to show the impact of the complexity of the programs on the proposed generator. We apply our proposed generator on larger programs, which are not limited to a single function. The generator can be applied to a main function that calls other functions.

Key contributions in this research paper are as follows:

- A classifier assigns the best optimization method for each condition. This can be extended to apply any optimization method that is deemed more efficient based on custom criteria. This classifier assigns the gradient descent method to single, continuous, and differentiable conditional statements, it assigns the particle swarm method to multiple, continuous and differentiable conditional statements, and it assigns the genetic algorithm to other discontinuous and non-differentiable conditional statements.

- The automatic test data generator is designed for Python programs. The application of the automatic test data generator has been extended to Python programs.
- The automatic process can handle not only a standalone function, but also nested functions within the main function. Previous works do not mention if nested functions can be handled.

Chapter 2

Theoretical Framework

This section discusses the theoretical background of this research paper. We will discuss the fundamental theories that support the validity of the proposed new method: the search-based software testing, dynamic execution and statistical pattern recognition. In this research paper, we utilize a combination of these theories to achieve a better solution for our problem.

2.1 Search-based software testing

The search-based software testing (SBST) has been the tool that is used to generate test suits for software systems. This method was first published by Webb Miller and David Spooner in 1976 [21]. Before this method was introduced, the symbolic execution and constraint solving was the mainstream technique for generating test data. The SBST involves executing the software, using a fitness function to evaluate the inputs, and searching for the best inputs that result in the lowest cost values using optimization algorithms. It mentions that the simplest optimization algorithm is random search. The next section discusses the fitness function.

2.2 Function minimization and objective function

In this section, we will discuss the objective function associated with the minimization problem that automatic test data generation transforms into.

In the first research of test data generation via dynamic execution [14], it introduces a way to organize the conditional statement in a form such that E1 op E2, where E1 and E2 are arithmetic expressions, and op is the comparison operator. In this early research, they assumed that there would be no logical operators such as AND or OR for simplicity. Later in [22], the conjunctions (AND) and the disjunctions (OR) are discussed. They proposed that in the presence of conjunctions, for example, for the conditional statement

if (A and B),

to satisfy the true branch of this condition, we only need to evaluate if B is true. This is because B is only evaluated if A is true due to the short circuit evaluation of the programming language. The tool in [22] is designed for C/C++ programs, but this feature is also used for Python.

For each single condition without conjunctions or disjunctions, the objective function is defined in Table 2.1.

	Condition	Objective function
1	x <= y	x - y
2	x == y	|x - y|
3	x >= y	y - x
4	Boolean (true/false)	if satisfied, 0; otherwise, 1000

Table 2.1: Objective functions for different types of conditions

In this research paper, we use a different approach for conjunctions and disjunctions. For disjunctions (OR), only the first term of the conditional statement is considered. For example, for the conditional statement

if (A or B),

we will only evaluate A, that is, convert A to its objective function based on Table 2.1, then solve this minimization problem to get the desired test case. For conjunctions (AND), all terms of the conditional statement are considered. For example, for the conditional statement

if (A and B),

we will evaluate both A and B, that is, convert A and B to their objective functions, respectively, based on Table 2.1, add the objective functions together, then solve this minimization problem to get the desired test case. This method also applies when there are more terms in a conditional statement. This way we perform the optimization on both conditions simultaneously.

2.3 Dynamic Execution

Dynamic analysis theory provides the foundation for searching for solutions while executing the programs. The existing automatic test data generation tool published in 2023 [18] did not utilize dynamic execution of programs. It has a restriction for attributes of objects that are assigned dynamically and that are difficult to identify upfront. In [15], variables that cannot be randomly initialized can be set up by utilizing Codex, which is an AI-powered tool that is trained on real-world coding tasks. However, the testing adequacy criteria might be limited, as it can only insert assert statements to generate the test data.

On the other hand, dynamic analysis executes the program, and the variable attributes are recorded. This lifts the limitations on unknown variable attributes, such as dynamically assigned types or special formats of a general type. The program is executed with randomly initialized input parameters of the main function, then all the other variables are recorded as the program executes.

2.4 Statistical pattern recognition

Anil discussed the pattern recognition techniques in 1987, that it assigns categories to different objects or events based on its observed patterns [11]. The design of the classifier is based on patterns observed from initial experiments.

Chapter 3

Optimization Algorithms

This research paper aims to explore different dynamic test data generation methods written in Python. We choose to explore Python because it has become the most popular and widely used programming language, especially in the data science and machine learning fields [17]. For dynamic test data generation, the values of local variables can be calculated dynamically and used to determine if the conditions are satisfied. To generate software test cases using optimization algorithms that cover the if statements of a program, the if statements are converted to objective functions. The optimization algorithms aim to minimize the objective value, so that the test case satisfies the statement. This section discusses the continuous objective functions. As test cases are generated, all conditions will be evaluated to see if the test case happen to satisfy any conditions.

3.1 Random Generation

A traditional way of automatic software testing is to generate random test cases to detect faults. Values are generated completely randomly for each parameter of the program. This method is used together with manual unit testing to catch random cases that might be skipped during manual testing. The effectiveness of this method usually depends on the number of test cases generated; therefore the computational costs is potentially high, especially when there are a significant number of parameters. This method minimizes the implementation costs upfront, but is not efficient when comparing the number of effective test cases generated to the total number of iterations, which may also increase the costs of storage of these test cases.

In this research paper, this method is used as a benchmark to evaluate the effectiveness of optimization algorithms and the proposed algorithm.

3.2 Direct approach – Least Squares

Given that the minimization problem can be written as a system of equations, one of the direct method of solving this problem is to solve the system of equations. The test cases are generated by directly solving the linear system Ax = b. To transform the automatic test data generation to a minimization problem, we use the objective functions in Table 2.1. Then we used the least squares method to solve the minimization problem. The least squares is a method that tries to find the optimal solution even when there is no exact solution [10]. In this research paper, we do not consider complex numbers. The method tries to solve this system:

$$\min_x ||Ax - b||_2,$$

where $A \in \mathbb{R}^{m \times n}$,$x \in \mathbb{R}^{n \times k}$, $b \in \mathbb{R}^{m \times k}$. The variable k represents the number of conditions that need to be met.

The program is first divided into various paths that it can take. For each path, all conditions that need to be satisfied to reach the end of the path are combined to form one of the main conditions that need to be satisfied for the condition coverage testing. Taking triangle classification as an example, the first condition is "if(($i <= 0$)or($j <= 0$)or($k <= 0$))". We only consider the first operand of the OR expression, which can be converted to

A = $\begin{bmatrix} A11 & 0 & 0 \end{bmatrix}$,

x = $\begin{bmatrix} x11 \\ x12 \\ x13 \end{bmatrix}$,

b = $\begin{bmatrix} -randnum \end{bmatrix}$.

The setup for the least squares problem for the whole triangle classification program is as follows:

A is 27 by 27 matrix and is set up as follows:

$$\begin{bmatrix}
A11 & A12 & A13 & 0 & 0 & 0 & 0 & 0 & 0 & 0... \\
0 & 0 & 0 & A24 & A25 & A26 & 0 & 0 & 0 & 0... \\
0 & 0 & 0 & A34 & A35 & A36 & 0 & 0 & 0 & 0... \\
0 & 0 & 0 & A44 & A45 & A46 & 0 & 0 & 0 & 0... \\
0 & 0 & 0 & 0 & 0 & 0 & A57 & A58 & A59 & 0... \\
0 & 0 & 0 & 0 & 0 & 0 & A67 & A68 & A69 & 0... \\
0 & 0 & 0 & 0 & 0 & 0 & A77 & A78 & A79 & 0... \\
... & ... & ... & ... & ... & ... & ... & ... & ... & ... \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0...
\end{bmatrix}$$

x is a 27 by 9 matrix and is set up as follows:

$$\begin{bmatrix}
x11 & 0 & 0 & 0... \\
x12 & 0 & 0 & 0... \\
x13 & 0 & 0 & 0... \\
0 & x21 & 0 & 0... \\
0 & x22 & 0 & 0... \\
0 & x23 & 0 & 0... \\
0 & 0 & x31 & 0... \\
0 & 0 & x32 & 0... \\
0 & 0 & x33 & 0... \\
... & ... & ... & ... \\
0 & 0 & 0 & 0...
\end{bmatrix}$$

b is a 27 by 9 matrix and is set up as follows:

$$\begin{bmatrix}
b11 & 0 & 0 & 0... \\
0 & b21 & 0 & 0... \\
0 & b22 & 0 & 0... \\
0 & b23 & 0 & 0... \\
0 & 0 & b31 & 0... \\
0 & 0 & b32 & 0... \\
0 & 0 & b33 & 0... \\
... & ... & ... & ... \\
0 & 0 & 0 & 0...
\end{bmatrix}$$

Take the triangle classification program as an example, there are 3 input variables needed. Elements in A for the first main condition are in the first 3 columns of A. The number of rows it takes depends on the number of small conditions that need to be satisfied simultaneously. For example, the first main condition is i <= 0, A11 = 1, A12 = 0, A13 = 0, b11 = -randnum. The second main condition is i = j > 0 and k > 0, A24(i) = 1, A25(j) = -1, A35(j) = 1, A46(k) = 1, b21 = 0, b22 = randnum, b23 = randnum. This process is repeated until we have coded all conditions in the format of Ax = b.

Finally, the least squares method is used to solve the linear system. This method has a runtime of 6.20e-05 second. A sample of test cases generated is as follows:

$$\begin{bmatrix}
-90 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 58 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 58 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 44 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 48 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 11 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 48 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 60 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -15 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -18 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 18 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 53 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 53 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 53 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 75 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 90 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 68 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 44 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 25 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 46 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 6 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 29 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 45
\end{bmatrix}$$

Although the least squares method is the most straight-forward method when we think of solving a linear system, it cannot be solved dynamically. As a result, it does not know the values of local variables. Similarly, it cannot handle functions calls, as these functions might be defined locally. Test cases generated by one conditional statement cannot be used to evaluate other conditions.

A turnaround for solving the first limitation mentioned earlier is to set up the matrix gradually while dynamically executing the program. However, as shown in Figure 3.1, as the matrix becomes larger, the runtime will significantly increase. It will be more efficient to perform an optimization algorithm to find the solution for the current condition instead of only setting up the matrix.

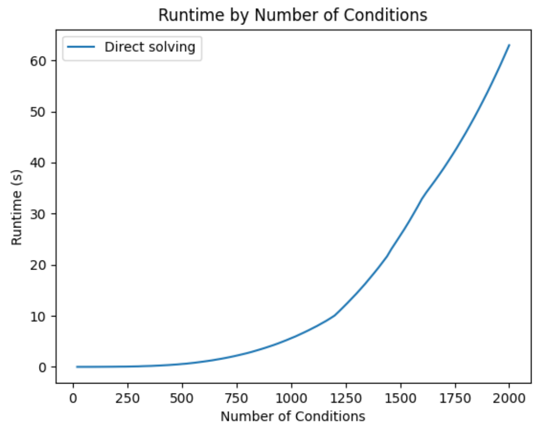


Figure 3.1: Comparison of Runtime for Different Number of Conditions Covered

Figure 3.1 shows runtime in seconds by number of conditions. The single value decomposition method is used to solve the least squares problem. It shows that there is a cubic relationship between runtime and the number of conditions. As number of conditions increases, the percentage increase in runtime continues to increase. We can conclude that, for real life programs, the least squares method will not be an efficient solution since the runtime increases cubically as the number of conditions increases.

3.3 Gradient descent optimization for differentiable conditions

In early research on automatic test data generation via dynamic execution [14], a similar method, the alternating variable method, was used to solve the minimization problem. The method adjusts each variable in turn towards the minimum.

Similarly, the gradient descent used in the context of test data generation also adjusts each variable in turn. This method requires the objective function to be continuous and differentiable. The algorithm uses the gradient of the objective function, and it gradually moves the variable toward the goal with the following algorithm:

$$x = x - \text{gradient} \times \text{step size}.$$

The limitations of this method are that it requires a gradient and it might be trapped at a local minimum. The step size chosen for this research is 1. This works fine with variables with small ranges. For further fine-tuning, we also explored accelerated gradient descent to dynamically calculate the step size, and gradient descent with line search to optimize the searching process.

For each single condition without conjunctions or disjunctions, the gradient is defined in Table 3.1 The gradients are only applicable for continuous and differentiable functions.

	Condition	Gradient
1	x <= y	1
2	x == y	1 if (x > y > 0); otherwise -1
3	x >=y	-1
4	Boolean	n/a

Table 3.1: Gradients for different types of conditions

3.4 Gradient descent optimization for non-differentiable conditions

Regular gradient descent cannot properly handle non-differentiable function, as there is no direct gradient. The objective function is not differentiable when the conditional is boolean, in other words, it is a true/false condition. In this case, the finite difference approximation can be used.

3.4.1 Finite difference approximation

When there is no direct gradient, 10 test cases are generated. The objective values are calculated for each test case, then an approximate gradient is calculated for each test case as follows [25, Chapter 8.1]:

$$\frac{\mathrm{d}f}{\mathrm{d}x_i} = \frac{f(x + h_i e_i) - f(x)}{h_i}$$

The average of these gradients is then used to update each parameter.

However, this still cannot easily handle function calls or loops dynamically, because for boolean conditions, the objective value equals a constant penalty, therefore, there will be no difference in the objective values for each test case. Instead, 10 random test cases are generated and the one with the lowest objective value will be used for the next iteration.

3.5 Accelerated gradient descent optimization

The following accelerated gradient (AGD) algorithm is used (Ghadimi et al., 2016) [7]:

1. Set
$$x_k^{md} = (1 - \alpha_k)x_{k-1}^{ag} + \alpha_k x_{k-1}$$
2. Compute $\nabla\Psi(x_k^{md})$ and set
$$x_k = x_{k-1} - \lambda_k \nabla\Psi(x_k^{md})$$
$$x_k^{ag} = x_k^{md} - \beta_k \nabla\Psi(x_k^{md})$$
3. Set k ← k + 1 and go to step 1

x_k, x_k^{ag}, x_k^{md} are initialized by the initial random test case, and will be updated with the AGD algorithm. k is the number of iterations, which starts at 1. $\alpha_1 = 1$ and $\alpha_k \in (0, 1)$ for any $k \geq 2$, $\beta_k > 0$, and $\lambda_k > 0$.

When a random test case is generated to take a different path of the program, x_k, x_k^{ag}, x_k^{md} are reset to the new test case because we need to include the values of the required local variables in the context of them.

3.6 Gradient descent optimization with line search

The following gradient descent with line search is used [1, Chapter 9.3]:

1. Set $\beta = 0.25$
2. During the iterations, if $f(x - t\nabla f(x)) > f(x) - \frac{t}{2}||\nabla f(x)||^2$. update t=$\beta$t
3. Use t as the step size for the gradient descent algorithm.

3.7 Particle swarm optimization

The particle swarm optimization method is inspired by the social behavior in nature that can be observed in herds of animals, flocks of birds, etc. [28]. This method introduces diversity in the test cases it generates, which increases the speed of finding the best solution. In addition, the PSO does not get trapped at local minimums easily because it searches around each local particle and also knows information from the overall population.

In the context of test data generation, for each condition, the number of dimensions for PSO is the number of variables. The following algorithm is followed:

1. Randomly generate a population of particles. Each particle consists of randomly generated values of the input parameters of pre-defined types.

2. If the condition has not been satisfied, update the velocity, which is used to adjust each particle. For each parameter $x_i^1, ..., x_i^d$, the velocity depends on both the personal best score/position ($pbest_{f_i(d)}^d$) and the global best score/position ($gbest_{f_i(d)}^d$) at the previous iteration, where i represents the number of iterations, $f_i(d)$ is the function that measures the score/position. The velocity is updated with the following formula [4] [28]:

   $$v_i^d(t) \leftarrow \omega \cdot v_i^d(t-1) + c \cdot r_i^d \cdot (pbest_{f_i(d)}^d(t-1) - x_i^d(t-1)) + c \cdot r_i^d \cdot (gbest_{f_i(d)}^d(t-1) - x_i^d(t-1)),$$

   where ω is the inertia weight which is set to 0.5, and c is the cognitive coefficient which is set to 1.5. r_i^d is a uniformly distributed random variable in the range of [0, 1].

3. Adjust the particles and store the personal best score/position and the global best score/position. The scores are the fitness values for each particle. The global best is the best particle that has the least fitness value in our minimization problem.

4. Repeat steps 2 and 3. Stop if the best individual that satisfies the condition has been generated or the maximum number of iterations (10,000) has been reached.

The population size is set to 50. Each input parameter that is required to generate is defined as a particle. If the input parameter has a higher dimension (e.g. list), each item counts as a separate particle. The fitness value is calculated by evaluating the value of the objective function as in Table 2.1 using the values of local variables.

3.8 Genetic algorithm

Using the genetic algorithm to solve the function minimization problem to find the desired test data is the main contribution in [22]. This method does not stop at a local minimum, and is helpful to solve multiple minimization functions simultaneously.

The genetic search algorithm mimics a natural selection process. In the context of test data generation, the following algorithm is followed:

1. Randomly generate a population. Each individual consists of randomly generated values of the input parameters of pre-defined types.
2. Select individuals from the population. The roulette-wheel selection method is used, which assigns individuals with lower fitness values a higher probability to be selected.
3. Crossover and mutate to generate the next generation.
4. Repeat steps 2 and 3. Stop if the best individual that satisfies the condition has been generated or the maximum number of iterations (10,000) has been reached.

The population size is also set to 50 to be comparable with PSO. Uniform mutation and two-point crossover are used. The number of iterations refers to the number of generations needed to find the solution. The fitness value is calculated by evaluating the value of the objective function as in Table 2.1 using the values of local variables.

Chapter 4

Pattern-based automated classifier

4.1 Automation

The abstract syntax tree (AST) module is used to perform a static analysis of the program being executed and to parse the source program. The optimization calculation is inserted before each conditional statement. The transformed program is then called to be executed.

We ask the user to pre-define the input parameter types in their source code. In Python, it is not required for users to pre-define the parameter types, but this step is needed to generate random initial test cases.

A coverage table is used, as used to track the progress of the generation process in [22] and [5]. This table tracks the decision-conditional coverage during the process. The coverage is tracked at each iteration to determine whether the current conditional statement is satisfied, other conditional statements that happen to be satisfied, or if the whole process can be terminated when the required coverage percentage is reached.

There is a test case storage that stores the final test cases generated that satisfy the conditional statements. When the true branch of a conditional statement turns from 'false' to 'true', we store the current local test case to the test case storage, which is populated at the end of the automation process.

At each iteration, the assigned optimization algorithm is applied to find the optimal test case at the current step. The program is then executed again to update the values of local variables. The evaluation of the output test data is done at the beginning of the next execution. As a result, when the conditional statements are evaluated, the local variable values have been properly calculated.

The evaluation function is also inserted through AST to the source code. When evaluating, the transformed program will be executed and the conditional statement will be marked as satisfied if it has been reached and satisfied. Especially at the first evaluation step, we will update the false branch of the coverage table for all conditional statement that cannot be reached by the current test data, or are evaluated as 'false'.

4.2 Sub-goal

The sub-goal method is used for conditional statements that are nested in a block [14]. For conditions nested within an if block, the test data generated needs to satisfy the current conditional statement and also ensures the current condition is feasible. For example, for the triangle classification program,

```
if (tri == 0):                                                  1
    if ((i + j <= k) or (j + k <= i) or (i + k <= j)):          2
        tri = 4                                                 3
```

when we reach line 2 and use the optimization algorithm to find a test case that satisfies this conditional statement, the conditional statement in line 1 becomes the constraint.

The constraints are not only used at the evaluation step, but also add penalty to the fitness values of the objective functions. Therefore, during the optimization process, the constraints also help steer the produced result toward the desired direction. However, since the gradient descent algorithm does not require the fitness calculation, the constraints can only be applied at the evaluation step.

4.3 Heuristics

A heuristic approach is used to only consider variables that affect the current conditional statement. For example, for the following current conditional statement in the binary search program, only one element of a list will make its TRUE/FALSE status change.

```
if (arr[mid] == target)
```

Then only item $arr[mid]$ is considered for the optimization algorithm to find a solution, not the entire list. This heuristic approach further improves efficiency, especially for the gradient descent and the particle swarm optimization algorithms that adjust each element individually.

4.4 Condition classifier

The reason we want to assign the optimal optimization algorithm by a classifier is because although the genetic search algorithm is a general algorithm that can solve all function minimization problems, it takes longer time, especially for simple problems when compared with other algorithms. Therefore, for single conditions that are continuous and differentiable, we believe that it would be the most efficient to find the solution using the gradient descent algorithm, while for multiple continuous and differentiable conditions, the PSO would work better. To support this idea, results from preliminary experiments can be found in Chapter 5.

In figure 4.1, the automated process is demonstrated. This process is ran five times to get average results for the coverage rate, run-time taken, and number of iterations required. We report the average results over multiple runs as the final result to eliminate the effect of random noise and increase the reliability of the results.

Each run starts with a randomly generated initial test data based on the predefined parameter types. When the program starts to execute, it decides whether the conditional statement it reaches is the conditional statement it is currently working on. If it is the conditional statement currently worked on, if it is not satisfied yet, the best-suited optimization algorithm will be applied. The best candidate for the current iteration will be produced. If the conditional statement is satisfied, the process will switch to the next conditional statement that has not been satisfied according to the condition coverage table. When a solution that satisfies a conditional statement is found, it will be recorded.

Afterwards, the execution of the program is continued, where it can either continue to work on the next if block or nested conditional statements, or return back to the beginning of the iteration. When the number of iteration on the current conditional statement exceeds a threshold (100), a random test data is generated and the conditional statement currently worked on will switch to the next one.

For optimization algorithms that require a population or particles to be generated initially, for example, the genetic algorithm and the particle swarm optimization, the population is compromised of individuals that have been recorded that can reach the conditional statement and other randomly generated individuals. This strategy has been utilized in many existing literature [22][12], and has been proven to make the process significantly more efficient.

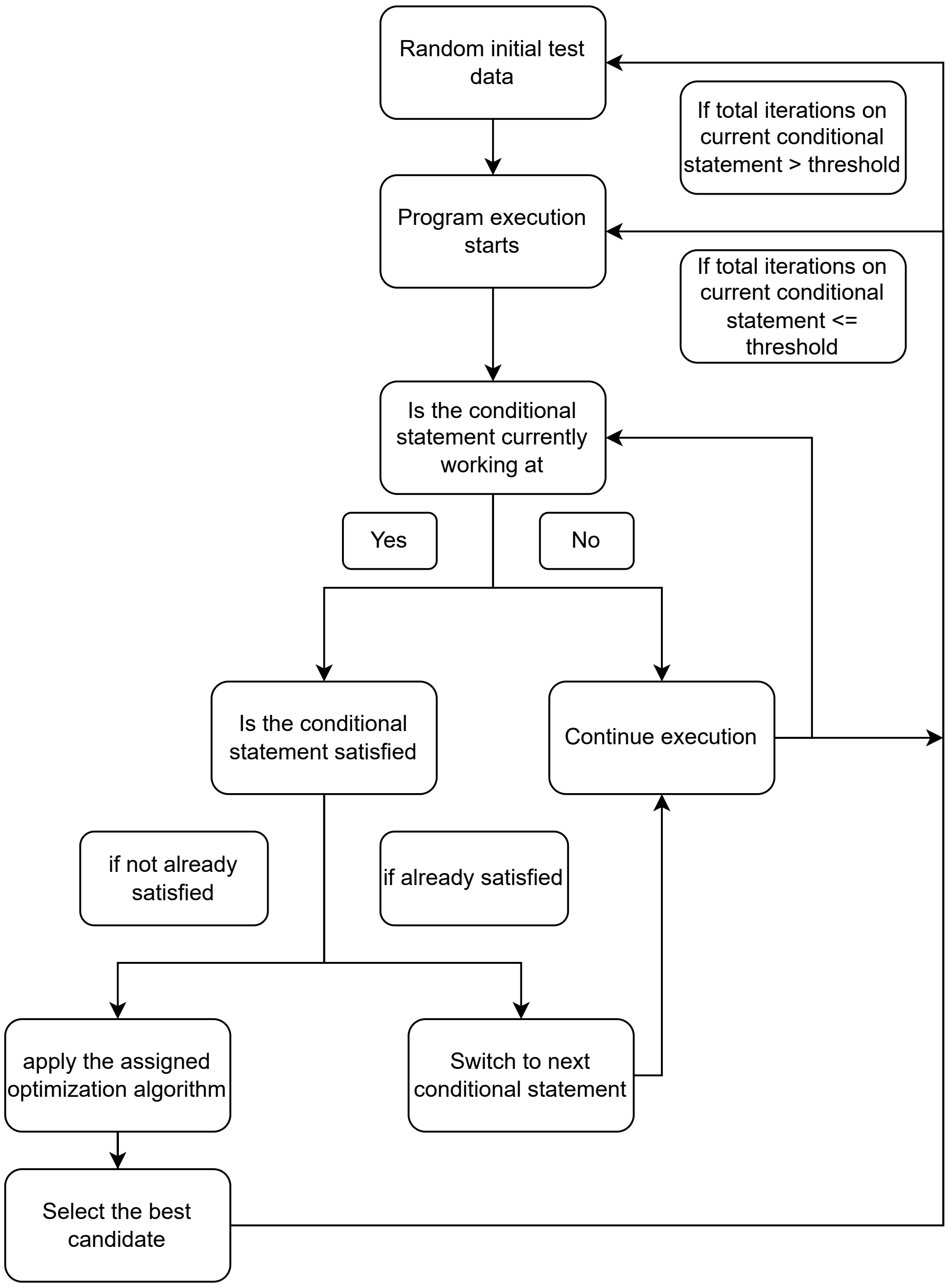

Random initial test data
If total iterations on current conditional statement > threshold
Program execution starts
If total iterations on current conditional statement <= threshold
Is the conditional statement currently working at
Yes
No
Is the conditional statement satisfied
Continue execution
if not already satisfied
if already satisfied
apply the assigned optimization algorithm
Switch to next conditional statement
Select the best candidate

Figure 4.1: Automation Process

Chapter 5

Experiment Results

5.1 Experiment design

The experiments are carried out on selected standalone programs using all algorithms described in Chapter 3: random, gradient descent, genetic algorithm, particle swarm optimization, and the classifier.

The following standalone programs are tested:

- binary search,
- triangle classification,
- analyze numbers. [6]

The binary search program has 4 conditional statement branches that are continuous and differentiable with only one single condition. The triangle classification program has 20 conditional statement branches. Some of them are continuous and differentiable with only one single condition; while others contain multiple continuous and differentiable conditions. The analyze numbers program has 22 conditional statement branches that are discontinuous and non-differentiable with both multiple conditions and single condition. The conditions are discontinuous because they are boolean conditions with function calls which are not differentiable and cannot slowly move to the destination in the direction of the steepest slope. They require dynamic evaluation because the values of function calls cannot be determined in advance. The first 2 programs are widely used in research papers to test the efficiency of the test case generation algorithm. To the best of our knowledge, programs similar to the analyze numbers program have not been used in recent research papers.

`analyze_numbers` is used as an example of programs that contain non-linear conditional statements. It contains function calls and loops that involve local variables and functions. This program takes in a list of integers and analyzes if the following conditions are satisfied:

Conditional Statements
If the list is empty
If all numbers are the same
If there are negative numbers
If there is at least 1 zero
If all numbers are even
If all numbers are odd
If all numbers are sorted in ascending order
If all numbers are sorted in descending order
If there are duplicate numbers
If some numbers are above average
If some numbers are below average

Table 5.1: `analyze_numbers`

We compared the coverage versus iterations between gradient descent and genetic algorithm. GA can reach a coverage of 91% while GD can only reach a coverage of 55%.

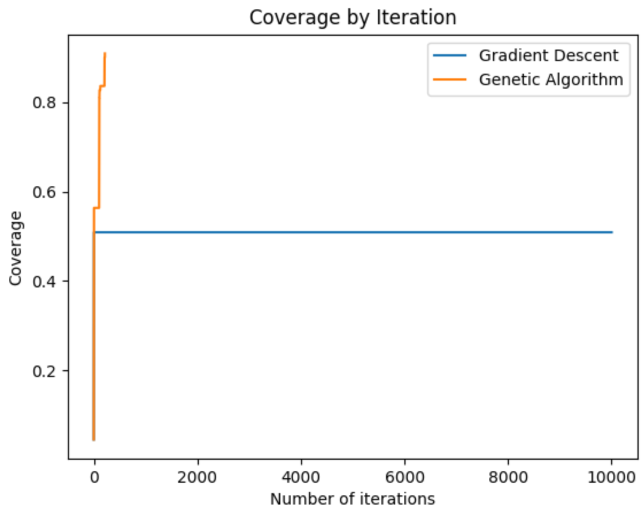


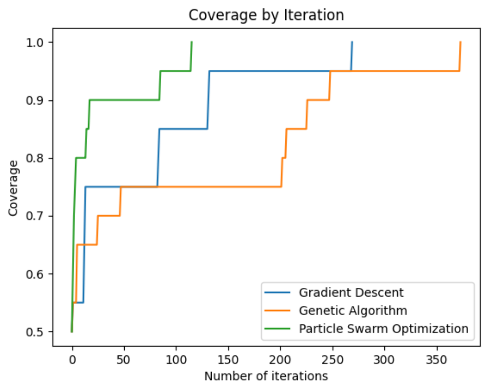
Figure 5.1: Comparison of Coverage vs Runtime for `analyze_numbers`

5.1.1 Effectiveness metric

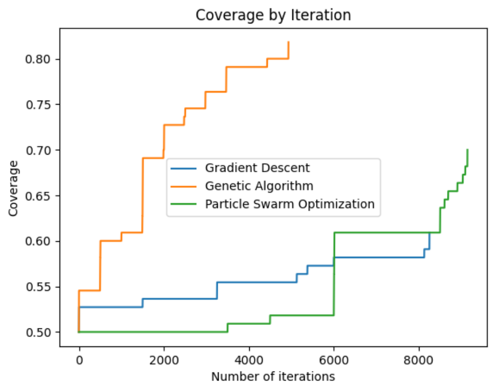
To measure the effectiveness of the algorithm, the condition-decision coverage is used as was used by Michael, McGraw, and Schatz (2001) [22]. A coverage table is created: the first column lists the conditional statements, the second column and the third column represents the true branch and the false branch respectively. Both true and false branches for all conditional statements default to "False". As test cases are generated for each conditional statement, the cell corresponding to the true branch of the conditional statement is marked "True" if the generated test case satisfies the conditional statement; otherwise its false branch is marked "True". The process continues until all conditional statements are satisfied or the number of maximum iterations is reached. The results shown in this section are the average taken over 5 iterations, to reduce the effect of random noises.

5.1.2 Number of iterations

The number of iterations used to reach the maximum coverage is compared among gradient descent, genetic algorithm, and particle swarm optimization. For each conditional statement, the process stops when the corresponding condition is satisfied or the total number of iterations has exceeded 10,000. The process moves on to the next nested conditional statement and iterates until the next conditional statement is satisfied.



(a) Continuous and Differentiable Conditions: Comparison of Coverage vs Iterations



(b) Discontinuous and Non-Differentiable Conditions: Comparison of Coverage vs Iterations

Figure 5.2: Comparison of Coverage vs Iterations

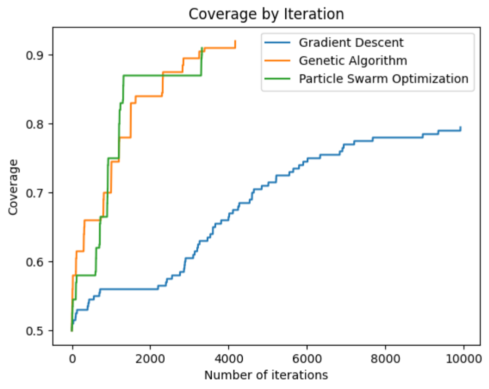


Figure 5.3: Conjunctions: Comparison of Coverage vs Iterations

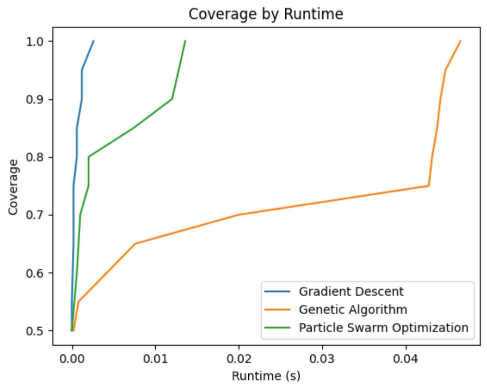
Figure 5.2a shows the coverage by number of iterations for a standalone program that contains only continuous and differentiable conditions. The three optimization algorithms can reach a coverage percentage of 100% before reaching the maximum iterations. On average, the gradient descent method requires 120 iterations to reach its best coverage for the binary search program which contains two conditional statements. Among the three algorithms, gradient descent performs the best as it requires the least number of iterations to satisfy all conditional statements.

Figure 5.2b shows the coverage by number of iterations for a standalone program that contains only discontinuous and non-differentiable conditions. The gradient descent algorithm only covered 62% of all conditional statement branches, the particle swarm optimization covered more at 70%, while the genetic algorithm found test cases that satisfy the most conditional statements at 82%. It can be observed that the genetic algorithm requires fewer iterations than the particle swarm optimization and gradient descent algorithm, while achieving a high coverage.

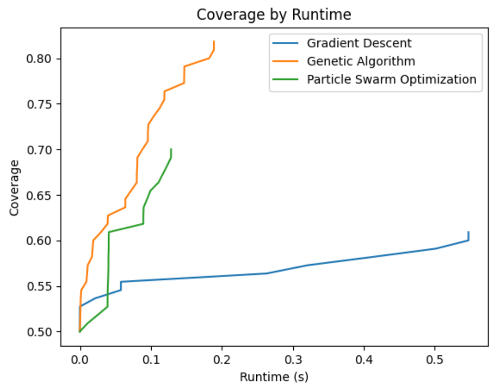
Figure 5.3 shows the coverage by number of iterations for a standalone program that contains conjunctions of continuous and differentiable conditions. The particle swarm optimization and the genetic algorithm achieved similar coverage, while the particle swarm optimization required relatively fewer iterations.

5.1.3 Runtime

In addition to the number of iterations, the runtime used was also compared. Some algorithms might need fewer iterations but take longer time to process.



(a) Continuous and Differentiable conditions: Comparison of Coverage vs Runtime



(b) Discontinuous and Non-Differentiable conditions: Comparison of Coverage vs Runtime

Figure 5.4: Non-Continuous and Non-Linear conditions: Comparison of Coverage vs Runtime

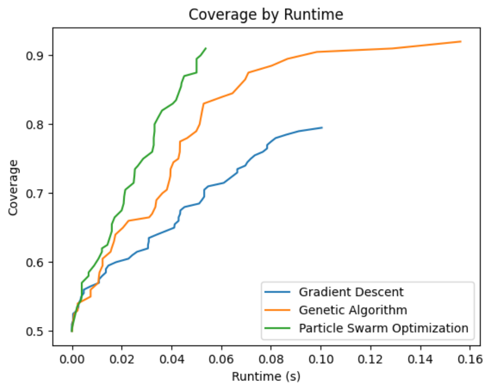


Figure 5.5: Conjunctions: Comparison of Coverage vs Iterations

Figure 5.4a shows the percentage of coverage by runtime for a standalone program that contains only continuous and differentiable conditions. Gradient descent also required the least runtime to finish its search, while the genetic algorithm required the most runtime.

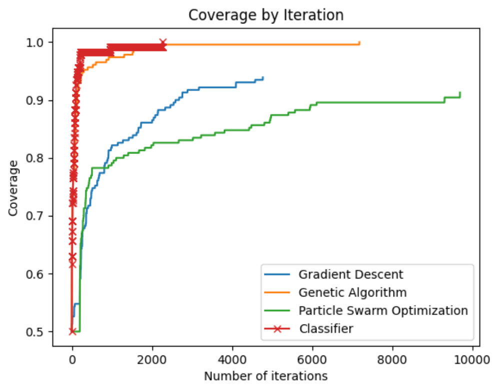
Figure 5.4b shows the coverage by runtime for a standalone program that contains only discontinuous and non-differentiable conditions. Both the genetic algorithm and the particle swam optimization took similar time; however, the genetic algorithm achieved a significantly higher coverage.

Figure 5.5 shows the coverage by runtime for a standalone program that contains conjunctions of continuous and differentiable conditions. The particle swarm optimization and the genetic algorithm achieved similar coverage, while the particle swarm optimization only required one thirds of the runtime for the genetic algorithm.

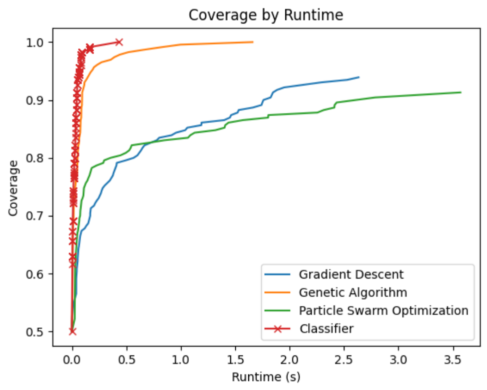
5.1.4 Mixes of both continuous and discontinuous conditions

From the above 2 metrics in section 5.1.2 and section 5.1.3, we can conclude that the gradient descent algorithm works the best for continuous and differentiable conditions, while the genetic algorithm works the best for discontinuous and non-differentiable conditions. In this section, the experiment is done on a more complex program that contains the above standalone functions as nested internally-called functions.

A classifier introduced in this research paper is used to generate test cases. For each condition, the classifier identifies whether the condition is continuous and differentiable or discontinuous and non-differentiable. The classifier also identifies if there are conjunctions of multiple conditions. The optimal optimization algorithm is assigned to each condition through a static analysis based on the patterns observed: for continuous and differentiable conditions, the gradient descent is applied, for discontinuous and non-differentiable conditions, the genetic algorithm is used, and for conjunctions of continuous and differentiable conditions, the particle swarm optimization is used.



(a) Mixed conditions: Comparison of Coverage vs Iterations



(b) Mixed conditions: Comparison of Coverage vs Runtime

Figure 5.6: Mixed conditions: Comparison of Coverage vs Runtime

Figure 5.6a shows the percentage of coverage by number of iterations for a complex program that contains continuous and discontinuous conditions and conjunctions.

Figure 5.6b shows the percentage of coverage by runtime for a complex program that contains continuous and discontinuous conditions and conjunctions.

It can be observed that the classifier helped to increase coverage while decreasing runtime.

5.2 Effect of Program Length

In this section, the effect of the complexity of programs in terms of number of conditions they contain is examined. The coverage and runtime are compared when the number of conditions is different. The results in this section helps to understand the scalability of the proposed method when it is applied to more complex problems.

5.2.1 Impact on Coverage

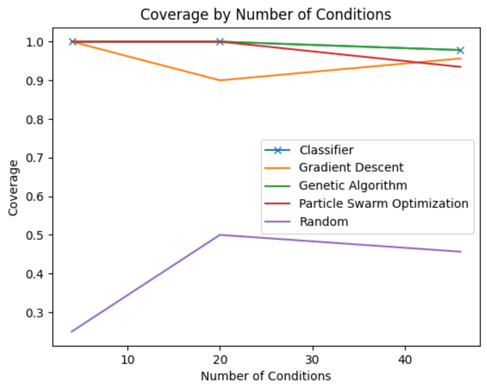


Figure 5.7: Classifier vs All methods: Comparison of Coverage for Different Number of Conditions Covered

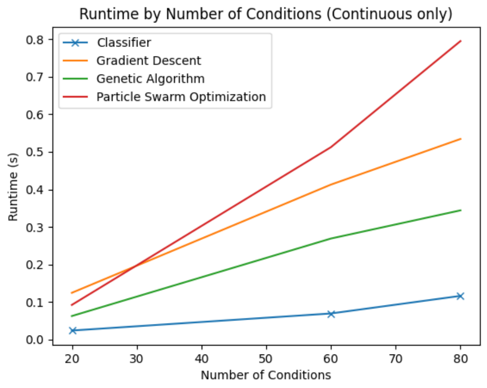
Figure 5.7 shows the percentage of coverage achieved for programs with different numbers of conditions. For number of conditions at 4, the programs contain only continuous and differentiable conditions. For number of conditions greater than 4 but less than 20, the programs contain only mixes of single and multiple continuous and differentiable conditions. For number of conditions more than 20, the programs contain both continuous and discontinuous conditions.

There is no significant effect on coverage for the classifier method when more conditions need to be covered, as highlighted in Figure 5.7. When number of conditions increases, the coverage is still approximately 100%. This is because the best optimization algorithm is being applied to each conditional statement individually, so that the coverage achieved will not be negatively affected, regardless of the type of the conditional statement.

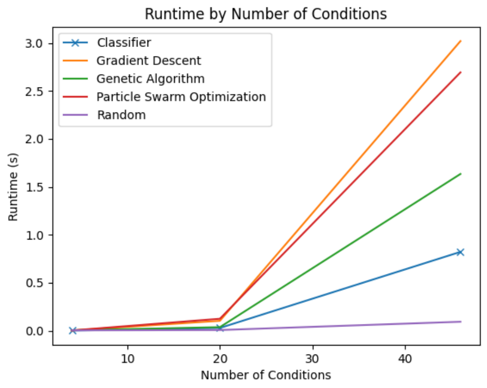
It can be seen that the random method only covered less than 50% of the conditional statement branches. This is used as a benchmark to show the effectiveness of the optimization algorithms.

5.2.2 Impact on Runtime

The runtime comparisons are conducted with a stopping point at a fixed condition-decision coverage percentage. The coverage threshold is used as the stopping criterion to ensure that runtime comparisons across all optimization algorithms are made under identical conditions.



(a) Comparison of Runtime (Continuous only)



(b) Comparison of Runtime (All Conditions)

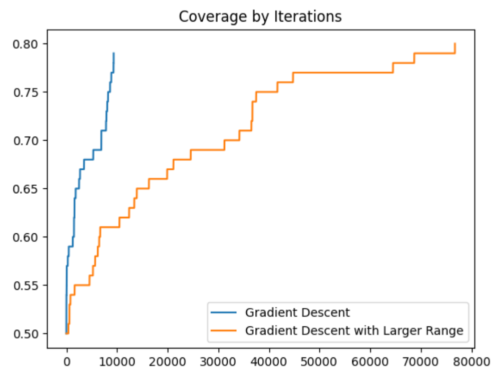
Figure 5.8: Comparison of Runtime for Different Number of Conditions Covered

Figure 5.8 shows the results in which we set the process to stop when the percentage of coverage reaches 85%. All of the genetic algorithm, the particle swarm optimization and the classifier can achieve a condition-decision coverage above 85%. When the number of conditions is relatively low, increasing number of conditions does not significantly impact runtime. However, as number of conditions increases, especially with discontinuous and non-differentiable conditions added, the runtime required increases significantly. However, it can be seen that the classifier requires the least runtime increase to reach the same percentage of coverage.

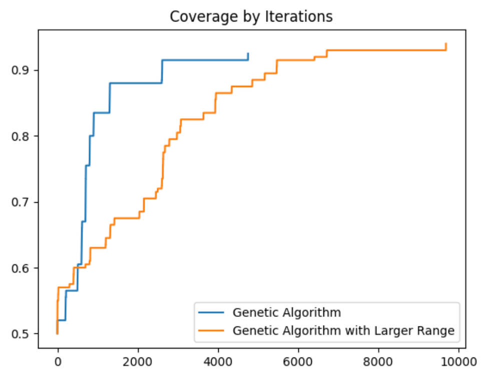
5.3 Effect of Range of Input Parameter Values

Compared with other papers, we see that the average coverage the model achieves matches, but the number of iterations required differs. This is possibly due to the different range that we assign for each variable. For our previous experiments, we assumed that the range for numbers is [-100, 100], and the range for the lengths of lists are [0, 10].

Figure 5.9 and Figure 5.10 show the coverage by iterations and coverage by runtime when we increase the range for numbers to [-1000, 1000] and increase the range for the lengths of lists to [0,100]. It can be observed that as the range of input parameters increases, the number of iterations and runtime required also increase.

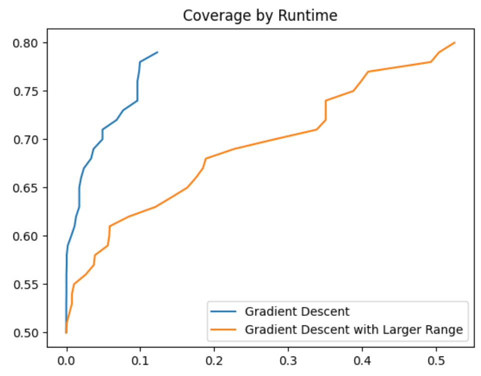


(a) Coverage vs Iterations for Different Ranges of Input Parameters

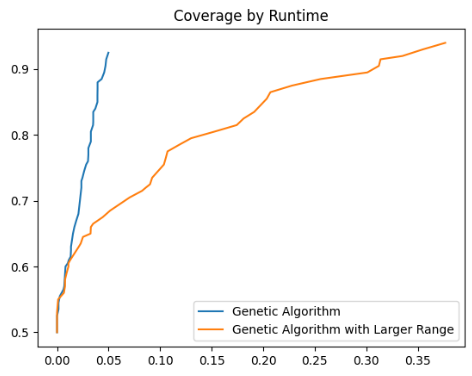


(b) Coverage vs Iterations for Different Ranges of Input Parameters

Figure 5.9: Coverage vs Iterations for Different Ranges of Input Parameters



(a) Coverage vs Runtime for Different Ranges of Input Parameters



(b) Coverage vs Runtime for Different Ranges of Input Parameters

Figure 5.10: Coverage vs Runtime for Different Ranges of Input Parameters

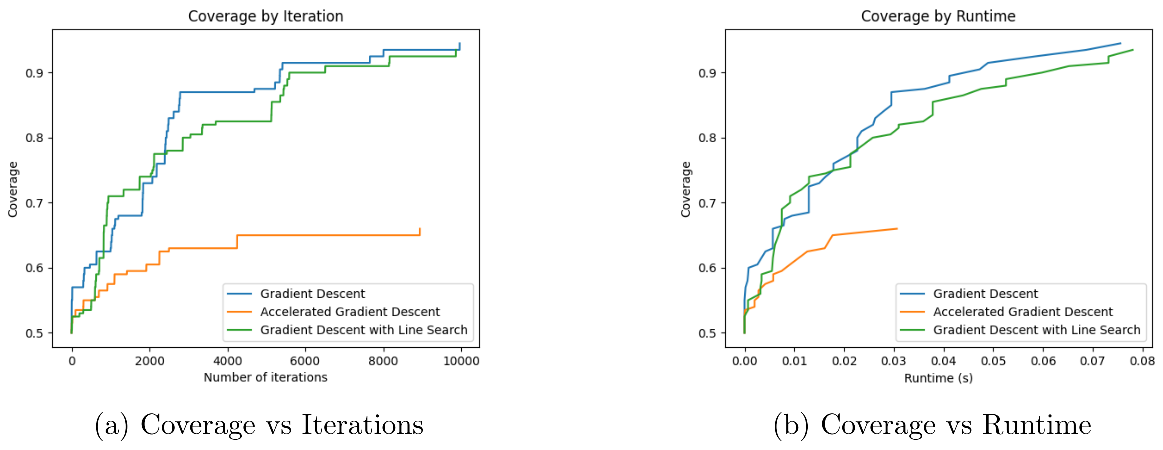
The range of input parameter values also affect the coverage. For example, for a input parameter that is a list, whether it can be an empty list matters. Therefore, the range of input parameter values must be adjusted before running the generation process.

5.4 Fine-Tuning

Fine-tuning involves adjusting the hyperparameters of each algorithm to find the most efficient setup.

5.4.1 Algorithm specific fine-tuning

For gradient descent, a fixed step size of 1, accelerated gradient descent and gradient descent with line search have been explored.



(a) Coverage vs Iterations (b) Coverage vs Runtime

Figure 5.11: Fine-tuning for Gradient Descent

5.4.2 General fine-tuning

The number of iterations required to loop through the automation process is set to be 10000.

The number of iterations required should be correlated with the range of input variables, as we observe in Figure 5.9 and Figure 5.10 to improve search efficiency.

For genetic algorithm and particle swarm optimization, a population of 50 is used. This can be further tuned for better performance.

For particle swarm optimization, a weighted inertia factor based on the number of iterations is used [12]. The cognitive coefficient is also adjusted based on the number of iterations. It starts with 1.5 and switches to 0.8 when the number of iterations reaches 50. This is to make it more precise when searching near the optimum.

5.5 Discussions

In this section, we examine various methods: the random generator, the least squares method, and three optimization algorithms: gradient descent, genetic algorithm, and particle swarm optimization. From our preliminary experiments, we found that the random generator cannot effectively generate test cases that cover all conditional statements with the same number of iterations or runtime compared to other methods. In addition, the gradient descent method works the best when the conditional statement contains one continuous and differentiable condition, but it cannot handle discontinuous conditions that involve function calls. The PSO is more efficient with conditional statements that contain conjunctions of continuous and differentiable conditions. The GA is more efficient with discontinuous and non-differentiable conditions.

From the preliminary results, the classifier assigns the most suitable optimization algorithm to each conditional statement. The results show that the classifier consistently reaches a coverage above 95% for all types of conditional statements, while requiring the least number of conditions and the least runtime.

Factors that negatively affect the runtime of the classifier method are the range of input parameters, the increase in the number of conditions, and the number of iterations spent on an inaccessible paths. Solutions that address these drawbacks can be explored in future work.

Chapter 6

Conclusions

This research paper aims to find a more efficient automatic test case generation method to increase the speed of the process and the conditional-decision coverage. We examined the coverage and speed for three types of optimization algorithms (Gradient Descent, Genetic Algorithm, and Particle Swarm Optimization) for different types of conditional statement and realized that there was no single algorithm that could perform the best for all types of conditional statement. Therefore, we proposed a classifier that assigns the best optimization algorithm to the current conditional statement by static analysis. It helped increase coverage and reduce runtime. Specifically, when running the program and generating test cases dynamically, the values of local variables are known and the best optimization algorithm can be chosen for each condition to achieve the best efficiency. This classification method can generally be applied to a broader selection of optimization algorithms and can be based on other custom criteria.

6.1 Limitations of the current approach

Limitations of the current classifier are as follows:

- If the parameter type is not built-in but is a user-defined class type, the current generator cannot properly generate test data for them. Some adjustments to the code base are required upfront.
- The current generator can only handle strings that are the identifiers of Enum members.

6.2 Future research ideas

There are future research ideas that can further improve the performance or overcome the limitations of the current classifier.

- The current tool can be further extended to be compatible with user-defined parameter types, so that it would automatically work without additional initial work.
- Other test adequacy criteria can be used to measure the effectiveness of the proposed classifier method.
- Other optimization algorithms, such as the ants colony, can be used to improve the classifier.
- Further fine-tuning for the optimization algorithms may further improve the results. For example, adjust the population size or the maximum number of iterations allowed before stopping.

References

[1] Stephen P. Boyd and Lieven. Vandenberghe. *Convex optimization.* Cambridge, Cambridge, UK ;, 2004.

[2] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select—a formal system for testing and debugging programs by symbolic execution. *SIGPLAN notices*, 10(6):234–245, 1975.

[3] L.A. Clarke. A system to generate test data and symbolically execute programs. *IEEE transactions on software engineering*, SE-2(3):215–222, 1976.

[4] Maurice. Clerc. *Particle swarm optimization.* ISTE ; v.93. ISTE, London ;, 2006.

[5] W.H. Deason, D.B. Brown, K.-H. Chang, and J.H. Cross. A rule-based software test data generator. *IEEE transactions on knowledge and data engineering*, 3(1):108–117, 1991.

[6] fumitoh, Matthew Caseres, Declan Naughton, and Open Source Modelling. lifelib. `https://github.com/lifelib-dev/lifelib`, 2025. Accessed: 2025-05-01.

[7] Saeed Ghadimi and Guanghui Lan. Accelerated gradient methods for nonconvex nonlinear and stochastic programming. *Mathematical programming*, 156(1-2):59–99, 2016.

[8] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE transactions on software engineering*, 36(2):226–247, 2010.

[9] W.E. Howden. Symbolic testing and the dissect symbolic evaluation system. *IEEE transactions on software engineering*, SE-3(4):266–278, 1977.

[10] Ilse C. F. Ipsen, Society for Industrial, and Applied Mathematics. *Numerical matrix analysis : linear systems and least squares*. Society for Industrial and Applied Mathematics SIAM, 3600 Market Street, Floor 6, Philadelphia, PA 19104, Philadelphia, Pa, 2009.

[11] Anil K. Jain, Pierre A Devijver, and Josef Kittler. Advances in statistical pattern recognition. In *Pattern Recognition Theory and Applications*, NATO ASI Series, pages 1–19. Springer Berlin / Heidelberg, Germany, 1987.

[12] Ya-Hui Jia, Wei-Neng Chen, Jun Zhang, Jing-Jing Li, Peter Whigham, Will N. Browne, Yuhui Shi, Grant Dick, Kay Chen Tan, Yaochu Jin, Lam Thu Bui, Xiaodong Li, Mengjie Zhang, Ke Tang, Hisao Ishibuchi, and Pramod Singh. Generating software test data by particle swarm optimization. In *Simulated Evolution and Learning*, Lecture Notes in Computer Science, pages 37–47. Springer International Publishing, Cham, 2014.

[13] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4. IEEE, 1995.

[14] B. Korel. Automated software test data generation. *IEEE transactions on software engineering*, 16(8):870–879, 1990.

[15] Caroline Lemieux, Jeevana Priya Inala, Shuvendu K. Lahiri, and Siddhartha Sen. Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models. In *Proceedings / International Conference on Software Engineering*, pages 919–931, Piscataway, NJ, USA, 2023. IEEE Press.

[16] Aiguo Li and Yanli Zhang. Automatic generating all-path test data of a program based on pso. In *2009 WRI World Congress on Software Engineering*, volume 4, pages 189–193. IEEE, 2009.

[17] Stephan Lukasczyk and Gordon Fraser. Pynguin: automated unit test generation for python. In *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 168–172, New York, NY, USA, 2022. ACM.

[18] Stephan Lukasczyk, Florian Kroiß, and Gordon Fraser. An empirical study of automated unit test generation for python. *Empirical software engineering : an international journal*, 28(2):36–, 2023.

[19] Stephan Lukasczyk, Florian Kroiß, Gordon Fraser, Aldeida Aleti, and Annibale Panichella. Automated unit test generation for python. In *Search-Based Software Engineering*, volume 12420 of *Lecture Notes in Computer Science*, pages 9–24. Springer International Publishing AG, Switzerland, 2020.

[20] M.R. Lyu. Software reliability engineering: A roadmap. In *Future of Software Engineering (FOSE '07)*, pages 153–170. IEEE, 2007.

[21] P. McMinn. Search-based software testing: Past, present and future. In *2011 Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163. IEEE, 2011.

[22] C.C. Michael, G. McGraw, and M.A. Schatz. Generating software test data by evolution. *IEEE transactions on software engineering*, 27(12):1085–1110, 2001.

[23] C.C. Michael, G.E. McGraw, M.A. Schatz, and C.C. Walton. Genetic algorithms for dynamic test data generation. In *Knowledge-Based Software Engineering Conference, 12th*, pages 307–308. IEEE, 1997.

[24] W. Miller and D.L. Spooner. Automatic generation of floating-point test data. *IEEE transactions on software engineering*, SE-2(3):223–226, 1976.

[25] Jorge. Nocedal and Stephen J. Wright. *Numerical optimization.* Springer series in operations research and financial engineering. Springer, New York, 2nd ed. edition, 2006.

[26] Anastasis A. Sofokleous and Andreas S. Andreou. Automatic, evolutionary test data generation for dynamic software testing. *The Journal of systems and software*, 81(11):1883–1898, 2008.

[27] Ke Wang, Yi Zhu, Guorui Li, Junjie Wang, Ziyang Liu, Nikolaos M. Freris, and Lei Chen. Test case generation method based on particle swarm optimization algorithm. volume 12721, pages 127211L–127211L–7. SPIE, 2023.

[28] Andreas Windisch, Stefan Wappler, and Joachim Wegener. Applying particle swarm optimization to software testing. In *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1121–1128, New York, NY, USA, 2007. ACM.