A Study on Improvement of Automated Test Input Generation with Machine Learning Techniques in Software Testing
(ソフトウェアテストにおける機械学習技術を用いた自動テスト入力生成の改善に関する研究)

Dissertation submitted in partial fulfillment for the
degree of Ph.D. of Advanced Science and Engineering

Xiujing Guo
郭　秀景

Under the supervision of
Professor Hiroyuki Okamura

Dependable Systems Laboratory,
Graduate School of Advanced Science and Engineering,
Hiroshima University, Higashi-Hiroshima, Japan

March 2024

Abstract

With the rapid increase of software scale and complexity, the cost of traditional software testing methods will increase faster than the scale of software. In order to improve test efficiency, it is particularly important to automatically generate high-quality test inputs. This article dissertation three approaches focusing on generating test inputs to achieve high test coverage (e.g. branch coverage) and generating test inputs based on boundary value analysis (BVA). We will introduce them successively.

In Chapter 2, we develop a framework for automatic test input generation based on the generative adversarial network (GAN). GAN is employed to train a generative model over execution path information to learn the behavior of the software. Then we can use the trained generative model to produce new test input, and select the test input that can improve the branch coverage according to our proposed selection strategy. Compared to prior work, our proposed method is able to handle programs under test with large-scale branches without analyzing branch expressions. In the experiment, we exhibit the performance of our method by using two modules in GNU Scientific Library. In particular, we consider the application of our method in two testing scenarios; unit testing and integration testing, and conduct a series of experiments to compare the performance of three types of GAN models. Results indicate that the WGAN-GP shows the best performance in our framework. Compared with the random testing method, the WGAN-GP based framework improves the test coverage of five functions out of the seven in the unit testing.

In Chapter 3, we focus on boundary value analysis (BVA). In software testing, a protective measure to prevent faults in the code is to ensure that the behavior on the boundary between the sub-domains of the input space is correct. Therefore, designing test inputs with boundary value analysis (BVA) can detect more errors and improve test efficiency. This dissertation presents an MLP (Multilayer Perceptron) based approach to automatically generate boundary test inputs. Our approach is twofold. First, we train an MLP-based discriminator that determines whether a boundary exists between two test inputs. Second, using the outputs of the discriminator, we create test inputs based on Markov Chain Monte Carlo. We conduct experiments to compare the fault detection capabilities of the MLP-based approach with concolic testing and manually-performed boundary analysis. Results indicate that the MLP-based method outperforms the manually-performed boundary analysis in four of the seven programs tested and concolic testing in three of the seven programs tested.

In Chapter 4, we discuss a boundary coverage metric for BVA. BVA is a common technique in software testing that uses input values that lie at the boundaries where significant changes in behavior are expected. This approach is widely recognized and used as a natural and effective strategy for testing software. Test coverage is one of the criteria to measure how much the software execution paths are covered by the set of test inputs. In this dissertation, we focus on evaluating test coverage with respect to BVA by defining a metric called Boundary Coverage Distance (BCD). The BCD metric measures the extent to which a test set covers the boundaries. In addition, based on BCD, we consider the optimal test input generation to minimize BCD under the random testing scheme. We propose three algorithms, each representing a different test input generation strategy, and evaluate their fault detection capabilities through experimental validation. The results indicate that the BCD-based approach has the potential to generate boundary values and improve the effectiveness of software testing.

Finally, we summary the contributions of this thesis are; (i) application of the GAN model to automatically generate test inputs, aiming to achieve full branch coverage.(ii) introduce an MLP based approach to automatically generate test inputs with BVA. (iii) propose a novel approach to evaluating test coverage in software testing, focusing on the utilization of Boundary Value Analysis (BVA) and introducing a new metric called Boundary Coverage Distance (BCD).

Acknowledgements

First and foremost, I would like to extend my sincere gratitude to Professor Hiroyuki Okamura, the supervisor of my study, for his valuable guidance, kind advice in every stage of the writing of this thesis. Without his continuous encouragement and impressive patience, I could not have completed my thesis.

Also, many thanks go to Professor Tadashi Dohi, Professor Hirashima Tsukasa, Professor Shaoying Liu and Professor Xiao Xiao for their invaluable comments, useful suggestions and warm encouragement.

Finally, it is my special pleasure to acknowledge the hospitality and encouragement of the past and present members of the Dependable Systems Laboratory, Graduate School of Advanced Science and Engineering, Hiroshima University. I also feel gratitude for my family's support and encouragement for my Ph.D. graduation.

Contents

Chapter 1

Introduction

Software testing is the execution of software systems for the purpose of revealing faults and is one of the most important activities in building a reliable software system. In general, software testing can be majorly classified into two categories; black-box testing and white-box testing. Black box testing is a testing technique where testers focus on the software's inputs and outputs without knowledge of its internal workings. Testers design test cases based on specifications and check whether the system's outputs meet expected outcomes. White box testing is a testing technique where testers have knowledge of the internal structure, design, and code of the software. Test cases are designed based on an understanding of the code logic to ensure all code branches are executed and produce expected results.

A common procedure of software testing is (i) to make test cases (The test case consists of test inputs and their expected outcomes by the test oracle.) (ii) to execute the software under test (SUT) with the inputs of test cases, and (iii) to compare the test outcomes with the expected ones. If the test outcome is different from the expected one, the SUT involves bugs. In dynamic testing, several topics need to be studied to improve testing efficiency, and they are roughly divided into three categories:

1. Test input generation: Test input generation focuses on creating diverse and effective test inputs to assess different aspects of a software system. In white box testing, the test inputs are generally designed to achieve a test coverage that attains a predefined level. In black box testing, test inputs are usually designed to cover different functional paths and bound-

ary conditions, such as boundary value analysis.In the process of testing, the software reliability may grow as the number of test inputs increases. However, a large number of test inputs cause a large amount of cost for software testing. Generating a test oracle for a large number of test inputs is a costly process, and executing extensive test input validation programs is also resource-intensive. Thus it is a challenge to consider how to reduce the number of test cases keeping a certain level of software reliability. The artificial intelligence (AI) is one of the key technologies to address this issue, and the AI-based software testing is expected to be a killer application for highly-reliable software development.

2. Test oracle generation: The test oracle is a mechanism to check whether the test result is correct or not. In the real situation, this is done by the expected test outcomes. However, since the expected test outcomes should be manually generated, it is the most costly process in software testing. There are several types of research results for the generation of the expected output with the artificial neural network (ANN) [1–4]. Valueian et al. [1] proposed a classifier-based method using ANNs that can build automated oracles for embedded software that has low observability and/or produces unstructured or semi-structured outputs. In the proposed approach, the oracles need input data tagged with two labels of "pass" and "fail" rather than outputs and any execution trace.

3. Test Metrics: Test metrics in software testing involve the measurement and analysis of various aspects of the testing process and the quality of the software under test. Metrics provide quantitative data that can help assess the progress, effectiveness, and reliability of the testing efforts. Test coverage is a crucial metric in software testing that gauges the extent to which a software application has been tested. It usually measures the percentage of code, branches, paths, requirements, or functionalities exercised by a set of test cases. In black box testing, boundary value analysis is a methodical approach to testing the boundaries of input domains, allowing testers to identify potential vulnerabilities and errors without detailed knowledge of the internal system logic. Boundary coverage is a critical aspect of test coverage that focuses on testing the software application

at or near its input boundaries. This is particularly important because boundary conditions are often sources of defects or unexpected behavior in software. However, identifying and testing all possible boundary conditions can be complex, especially in systems with intricate input domains.

In this dissertation, we first focus on generating test inputs to achieve full branch coverage. The test coverage is defined as a fraction of executed test path in testing over the entire execution paths of SUT [5]. In general, since it is difficult to measure the entire execution paths, statement and branch coverage are used instead of path coverage in practice. The search-based software testing (SBST) is a method to generate test inputs with search techniques, e.g., meta-heuristic optimization methods, so that it can achieve high coverage [6]. Fraser and Arcuri [7] presented an extension of SBST that is able to exercise automated oracles and to produce high coverage test suites at the same time. Zhan and Clark [8] applied search techniques to test input generation for Simulink models. In this approach, they proposed a full test-set generation framework, which successfully combines random testing with the search-based targeted test-input generation techniques to enable us to generate effective test sets. In general, to achieve the branch coverage [9], the SBST uses symbolic execution to extract branch conditions and uses an optimization algorithm to find the test input that satisfies the branch condition. However, with the increasing scale and complexity of the software system, it is complicated and time-consuming to carry out large-scale branch coverage. Therefore, we propose a software testing framework with Generative Adversarial Networks (GANs) to automatically generate high-coverage test inputs.

Secondly, we focus on automatically generate test inputs with BVA based on Multilayer Perceptron (MLP). Boundary value analysis (BVA) is one of the most popular methods to create the high-quality test cases effectively. The boundary value is defined as an input of software that changes the behavior of software with even a little change, and BVA extracts test inputs from the boundary values. In the context of black-box testing, BVA extracts the test inputs from the boundaries of equivalence partitions. In the BVA with white-box testing, the test inputs can be determined from branch and loop conditions on source codes. It is empirically known that many programming errors often occur on the boundary of the input domain. One protective measure to prevent vulnerabilities and failures in the code is to ensure correct behavior on the boundaries between the input space sub-domains [10]. Therefore, compared with other methods, by designing test cases with BVA, more boundary errors could be detected and the test efficiency is higher. On the other hand, since BVA requires the analysis of specifications or source codes, the effort of BVA is not small. In the case of black-box testing, testers identify the equivalence partitions or subdomains by analyzing the specification using partition analysis (PA), and create test inputs from the boundary between sub-domains [11]. This process generally relies on manual analysis. Since the complexity of the software system increases, software has large input spaces, heavily or non-linearly dependent inputs, and complex and highly structured inputs. Thus it is not feasible to manually analyze the input domain and the boundary values [10]. In recent years, there are several researches on the automation of BVA. Jeng and Forgacs [12] proposed a semi-automatic method that mixed the dynamic search method and the algebraic manipulations of the boundary conditions to generate test input for boundary value testing (BVT) more efficiently. Zhao et al. [13] considered string inputs and proposed a novel approach for automatically generating test points to better find problems at borders in code with string predicates. Ali S et al. [14] extended their search-based test input generation method in model-based testing, using a solver to automatically generate boundary values based on a set of heuristics. Feldt and Dobslaw [15] applied the idea of derivative in mathematical parlance to detect the maximum "change" area by combining the input and output distances, that is, the detection boundary. This method uses the program derivatives as a fitness function in search-based software testing for automated BVA. In order to generate boundary test cases through the above techniques, it is necessary to have the specification that clearly states the boundary formally. In this work, we consider the BVA in white-box testing. The BVA in white-box testing focuses on a pair of input and its execution path, and the boundary is defined as the input that changes the execution path in some sense[1]. Compared to the BVA in black-box testing, one of the difficulties of BVA in white-box testing is to identify feasible test inputs. On BVA in black-

[1]The definition is formally given in Chapter 3.

box testing, we implicitly suppose that the input variables on specifications are independent, and then the boundary can be obtained from combinations of the boundary values for each input variables. On the other hand, input variables are dependent; for example, the program includes a condition

```
    x + y <= 10
```

for two input variables `x` and `y`. Then the boundary of this condition cannot be determined only by either of `x` and `y`. That is, even if the boundary is detected from source codes, we need to solve the problem of finding feasible input values. For this problem, Zhang et al. [16] used the SMT (satisfiability modulo theories) solver. However, SMT has weaknesses in scalability. In other words, it is difficult to apply the SMT-based approach to the large-sized programs. Therefore, we propose another approach for the BVA in white-box testing, i.e., MLP based approach.

Thirdly, we propose a boundary coverage metric for BVA. BVA is a systematic testing technique used in software testing to design test cases that focus on the boundaries of input domains. The primary goal of BVA is to identify potential defects or errors that may occur at or near the edges or limits of the input space. To make BVA more effective, proposing a boundary coverage metric is important. By defining boundary coverage metrics, we can measure how thoroughly we test these critical areas and evaluate the quality of the test suite and ensure we don't miss critical test scenarios. Moreover, BVA is particularly useful for uncovering off-by-one errors, boundary-related exceptions, and other issues that often escape less focused testing. A boundary coverage metric allows to quantify the coverage of these high-risk areas, reducing the chances of releasing software with critical defects. At the same time, testing can be time-consuming and resource-intensive. By establishing boundary coverage metrics, we can prioritize testing efforts. Focusing on achieving full coverage of the boundary during testing while conducting less exhaustive testing in non-boundary areas helps optimize resource allocation and testing efficiency. In summary, proposing a boundary coverage metric for boundary value analysis in software testing is essential for enhancing the precision, efficiency, and effectiveness of the testing process. In this work, we attempt to define boundary coverage measures, called boundary coverage distance (BCD).

The organization of this dissertation is as follows. The automated software test input generation framework with GAN model is shown in Chapter 2. Chapter 3 introduces the details of test suite generation with MLP-based boundary value analysis. Chapter 4 describes the content and application of boundary coverage distance. Chapter 5 gives the conclusions and the future work.

Chapter 2

Test Input Generation with Generative Adversarial Networks

2.1 Introduction

In this chapter, we propose a software testing framework with generative adversarial networks (GANs). Concretely, we consider a GAN model to generate test input for the SUT, and a test strategy to increase the test coverage with the GAN. The GAN is a NN model for the input generation from given training data. A GAN consists of a generator and a discriminator, and the most important feature of GAN is to reinforce the ability of data generation for the generator by competing to the discriminator to detect fake data [17]. The main contributions of this chapter are summarized as follows:

1. We propose a GAN model to generate test inputs and their associated execution paths simultaneously. That is, in our model, the discriminator learns program execution paths for SUT and the generator generates test cases including test inputs and expected paths.

2. We discuss a test input selection strategy to increase the test coverage.

3. We investigate the applicability of our framework for two software testing scenarios; unit testing and integration testing. The experimental comparison proves that our method is better than the common random test.

2.2 GAN: Generative Adversarial Network

In this section, we first introduce the GAN model and then present the input and output of GAN in our framework.

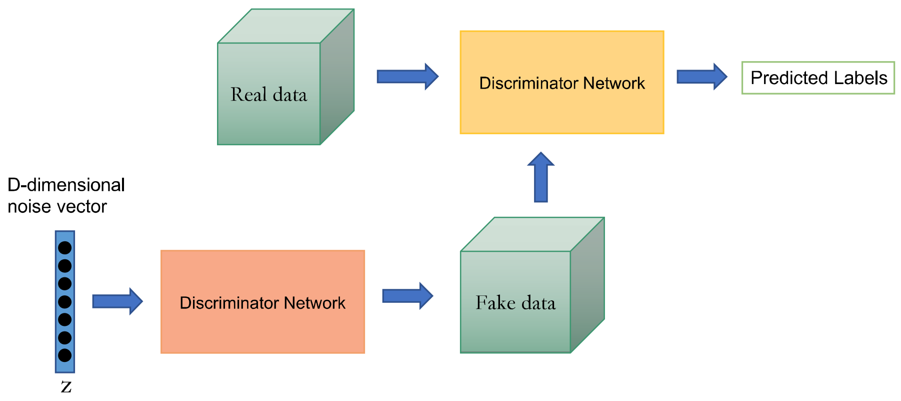
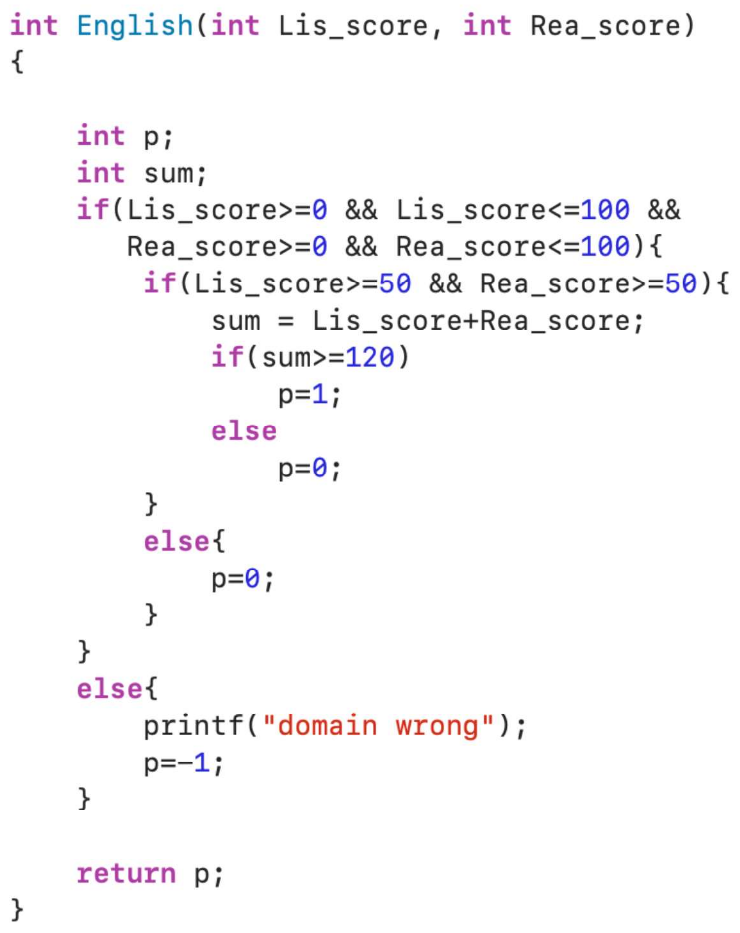


Figure 2.1: The architecture of GAN.

GANs are algorithmic architectures that use two neural networks, pitting one against the other (thus the “adversarial”) in order to learn the underlying distribution of the training data so that it can generate new data instances that resemble the training data [18]. Fig. 2.1 is the architecture of GAN. It consists of a generator and a discriminator. The generator takes random noise z and generates fake data. The discriminator determines if the generated data are real or fake. GAN estimates generative models via an adversarial process. In the adversarial process, the generator aims to maximize the failure rate of the discriminator while the discriminator aims to minimize it. The GAN model converges when the Nash equilibrium is reached [17].

One of the most challenging problems when AI technologies are applied to unstructured data such as computer program and software testing is to determine what data are used as inputs of the AI system. Our main idea is to use not only the test inputs but also the corresponding execution paths as inputs of GAN. A pair of test inputs and program execution paths includes rich information on the program itself. Also, it is not difficult to observe and collect the execution path for an input in the dynamic testing and it is more reasonable in terms of cost than symbolic execution used in the SBST.

In this chapter, we focus on branch coverage. Therefore, the execution path



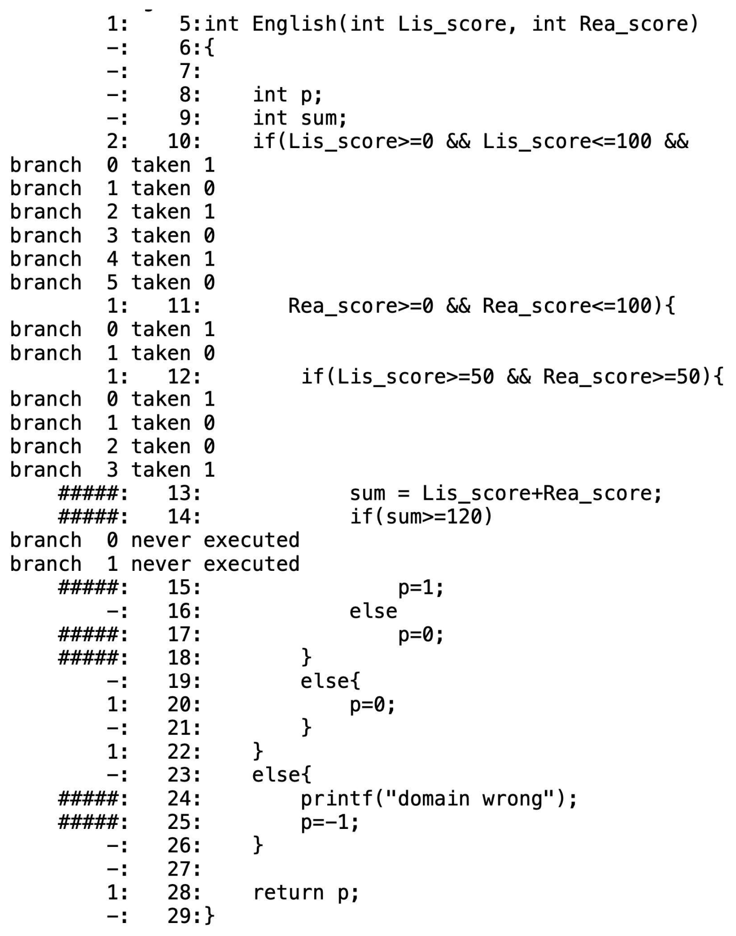
```c
int English(int Lis_score, int Rea_score)
{

    int p;
    int sum;
    if(Lis_score>=0 && Lis_score<=100 &&
       Rea_score>=0 && Rea_score<=100){
        if(Lis_score>=50 && Rea_score>=50){
            sum = Lis_score+Rea_score;
            if(sum>=120)
                p=1;
            else
                p=0;
        }
        else{
            p=0;
        }
    }
    else{
        printf("domain wrong");
        p=-1;
    }

    return p;
}
```

Figure 2.2: The source code of `En_testing.c`.

in our work is extracted by Gcov tool [19] and is defined as the combination of the execution of each branch in the program under a specific input. Gcov is a source code coverage analysis and statement-by-statement profiling tool that can generate exact counts of the number of times each statement in a program is executed, and that annotates source code to add instrumentation. Gcov is released with GCC and cooperates with GCC to realize statement coverage and branch coverage testing of C/C++ files. When executing a program, we use Gcov to write branch frequencies to the output file, and then extract the branch information in the file as a path. For example, Figure 2.2 is a simple C program for judging whether the English examination passed. We assume that the English examination consists of two parts; listening and reading. The full



```
              1:    5:int English(int Lis_score, int Rea_score)
              -:    6:{
              -:    7:
              -:    8:    int p;
              -:    9:    int sum;
              2:   10:    if(Lis_score>=0 && Lis_score<=100 &&
branch  0 taken 1
branch  1 taken 0
branch  2 taken 1
branch  3 taken 0
branch  4 taken 1
branch  5 taken 0
              1:   11:       Rea_score>=0 && Rea_score<=100){
branch  0 taken 1
branch  1 taken 0
              1:   12:        if(Lis_score>=50 && Rea_score>=50){
branch  0 taken 1
branch  1 taken 0
branch  2 taken 0
branch  3 taken 1
          #####:   13:            sum = Lis_score+Rea_score;
          #####:   14:            if(sum>=120)
branch  0 never executed
branch  1 never executed
          #####:   15:                p=1;
              -:   16:            else
          #####:   17:                p=0;
          #####:   18:        }
              -:   19:        else{
              1:   20:            p=0;
              -:   21:        }
              1:   22:    }
              -:   23:    else{
          #####:   24:        printf("domain wrong");
          #####:   25:        p=-1;
              -:   26:    }
              -:   27:
              1:   28:    return p;
              -:   29:}
```

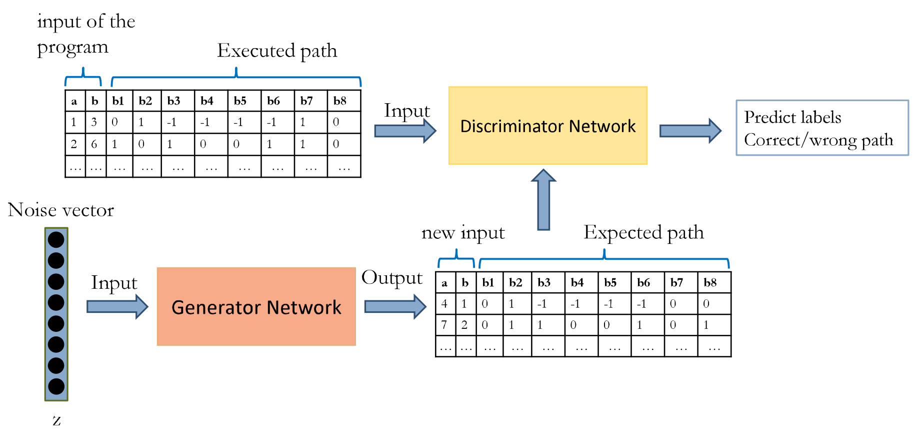
Figure 2.3: The result of Gcov for `En_testing.c.gcov`.

score of each part is 100 points. To pass the English examination, the following two conditions must be satisfied:

(1) Both the listening score a and the reading score b are greater than or equal to 50 points.

(2) The sum of listening score a and reading score b is greater than or equal to 120 points.

Figure 2.3 shows the result of Gcov on running the `En_testing.c.gcov` program with the input $a = 45$ and $b = 65$. We convert the "`taken a (a>0)`" to "`1`" to represent that the branch is taken at least once, convert the "`taken 0`" to "`0`" to represent that the branch is not taken, and convert the "`never executed`" to "`-1`" to represent that the branch is not executed, and the execu-



a	b	b1	b2	b3	b4	b5	b6	b7	b8
1	3	0	1	-1	-1	-1	-1	1	0
2	6	1	0	1	0	0	1	1	0
…	…	…	…	…	…	…	…	…	…

a	b	b1	b2	b3	b4	b5	b6	b7	b8
4	1	0	1	-1	-1	-1	-1	0	0
7	2	0	1	1	0	0	1	0	1
…	…	…	…	…	…	…	…	…	…

Figure 2.4: The input and output of GAN in our framework (”b1=0” represents that branch1 is not taken; ”b2=1” represents that branch2 is taken at least once; ”b3=-1” represents that branch3 is not executed).

tion path of the program `En_testing.c` with the input $a = 45$ and $b = 65$ is the combination of all branch executions; `1,0,1,0,1,0,1,0,0,1,-1,-1,-1,-1`.

Fig. 2.4 illustrates the input and output of GAN in our framework. The input of GAN is a noise vector and a set of inputs of the program with their corresponding executed paths extracted by Gcov. The output of GAN is a set of new inputs of the program with their corresponding expected paths generated by generator. In the successfully trained GAN model, the generator can generate the correct path for a specific generated input, and the discriminator can accurately determine whether the path corresponding to the specific input is correct. Therefore, the generator can be used to generate test inputs and improve test coverage based on generated path information, and the discriminator can be used as an evaluator.

2.3 Software Testing with GAN

We propose a framework for automated test data generation and aim to achieve the full branch coverage in software testing. In our framework, we use GAN to iteratively generate test inputs for software testing. Fig. 2.5 illustrates the structure of the framework. It contains the following four steps:

Step 1: Select m test inputs and their corresponding paths as training data.

Step 2: Train GAN with training data to generate n test inputs.

Step 3: According to the path information of the generated test inputs, select $w (w < n)$ test inputs from n test inputs sets that may cover not-executed branches in the training data.

Step 4: Add the selected w test inputs and their corresponding executed paths to the training data. Go to Step 2.

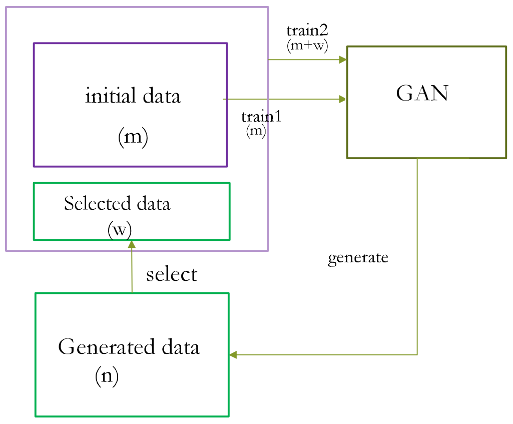
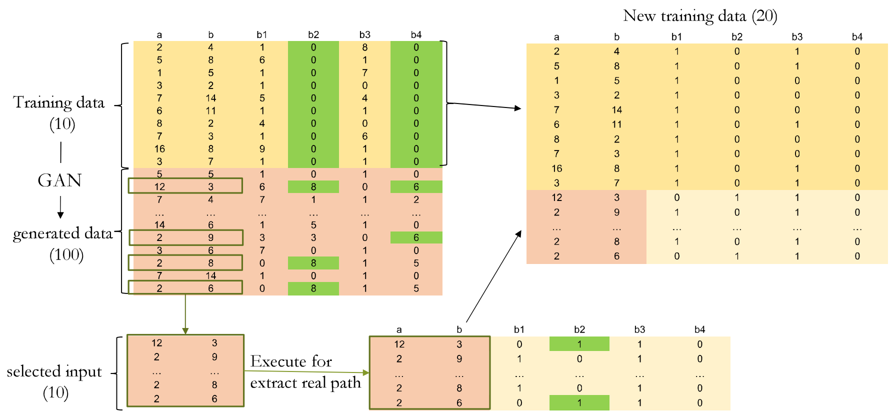


Figure 2.5: A framework for the test input generation indenting to increase test coverage.



a	b	b1	b2	b3	b4
2	4	1	0	8	0
5	8	6	0	1	0
1	5	1	0	7	0
3	2	1	0	0	0
7	14	5	0	4	0
6	11	1	0	1	0
8	2	4	0	0	0
7	3	1	0	6	0
16	8	9	0	1	0
3	7	1	0	1	0
5	5	1	0	1	0
12	3	6	8	0	6
7	4	7	1	1	2
...	...	...	...	...	...
14	6	1	5	1	0
2	9	3	3	0	6
3	6	7	0	1	0
2	8	0	8	1	5
7	14	1	0	1	0
2	6	0	8	1	5

12	3
2	9
...	...
2	8
2	6

a	b	b1	b2	b3	b4
12	3	0	1	1	0
2	9	1	0	1	0
...	...	...	...	...	...
2	8	1	0	1	0
2	6	0	1	1	0

New training data (20)

a	b	b1	b2	b3	b4
2	4	1	0	1	0
5	8	1	0	1	0
1	5	1	0	0	0
3	2	1	0	0	0
7	14	1	0	0	0
6	11	1	0	1	0
8	2	1	0	0	0
7	3	1	0	0	0
16	8	1	0	1	0
3	7	1	0	1	0
12	3	0	1	1	0
2	9	1	0	1	0
...	...	...	...	...	...
2	8	1	0	1	0
2	6	0	1	1	0

Figure 2.6: An example of proposed framework.

Fig. 2.6 illustrate an example of our framework. In the training data, branch 2 and branch 4 are not taken. Therefore, we select the inputs "a=12, b=3; ... ; a=2, b=9; a=2, b=8; a=2, b=6" where generated path information shows that branch 2 and branch 4 are taken. Then execute the program with selected inputs to extract real path, add them into the training data. With a high-precision GAN model, our framework has the ability to improve test coverage.

2.4 Experiment

In this section, we conduct a series of experiments to evaluate the effectiveness of the proposed framework. First, we applied our framework to unit testing and integration testing, and next selected three GAN models: WGAN-GP [20], BiGAN [21], and standard GAN [17] in our experiments. Our experiments aim to evaluate which type of GAN is more suitable for the framework, and which test level is suitable for our framework.

2.4.1 Experimental setup

In the past few years, different variants of GAN have been proposed. In our experiments, we choose three GAN models to apply to our framework: WGAN-GP, BiGAN, and the standard GAN. WGAN-GP is an improvement of WGAN [22] with a gradient norm penalty. The main difference from the standard GAN is only the cost function. WGAN-GP is based on the Wasserstein distance that has a smoother gradient everywhere. Therefore it has the higher training stability than the standard GAN [20]. BiGAN is a type of GAN where the generator not only maps latent samples to generated data, but also has an inverse mapping from data to the latent representation [21]. We compared the test coverage of the test input generated by the three GAN models and the accuracy of the generated path.

We apply our proposed framework to two software testing levels: unit testing and integration testing, conduct two sets of experiments on WGAN-GP, BiGAN, and the standard GAN. Unit testing is a type of software testing in which individual units or components of the software are tested. The purpose is to verify that each unit of the software code executes as expected. A unit can be a single function, method, procedure, module, or object. Integration testing is a level of software testing in which individual units/components are combined and tested as a whole. The purpose of this test is to expose errors in the interaction between integration units. Integration testing can be performed by the following strategies: incremental approach and big bang approach. In the incremental test method, the test is performed by integrating two or more logically related modules, and then the function of the application is tested. Then, other related modules are integrated incrementally, and this process is repeated until all logic-related modules are successfully integrated and tested. Big bang integration test is an integration test strategy that combines all the units at once to form a complete system, and then tests the unity of different units as one entity. In our unit test experiment, the GAN model generates paths in function units. In the integration test experiment, we adopt the big bang strategy and use GAN to generate paths for all functions at once.

In both experiments, the parameters of the framework are set to $m = 100$, $n = 100$, $w = 10$, and are iterated 10 times to generate 100 test inputs. In the integration testing, the 100 initial data are randomly generated. In the unit testing, we do not directly consider the input parameters of a single function, but consider the input of the main function. Therefore, the initial data of each function in unit testing is the selected test inputs that can pass the objective function. Regarding the network structure of the generator and discriminator in the GAN model, we applied a simple fully connected neural network (dense layers) and a convolutional neural network for comparison. The GAN model trains 5000 epochs per iteration in the unit testing, and 10000 epochs per iteration in the integration testing (one epoch is the cycle when an entire dataset is passed forward and backward through the neural network only once.) In the BiGAN models, the encoder model is an inverse mapping from data to the latent representation. Since the authors of the paper [20] recommend to use layer normalization as a drop-in replacement for batch normalization in WGAN-GP to help stabilize training. The same authors also use the RMSProp optimizer in their experiment. Therefore, we also use the RMSProp optimizer and layer normalization in WGAN-GP, and use the Adam optimizer and batch normalization in other GAN models. During the training process, it's important to monitor the changes in the gradient and loss of the network and balance the training of the Generator and Discriminator. Ideally, the generator should receive a large gradient early in training, as it needs to learn how to generate realistic data. On the other hand, the Discriminator should not always receive a large gradient

Table 2.1: The functions names.

gamma_inc.c	**Function1**	gsl_sf_gamma_inc_Q_e
	Function2	gsl_sf_gamma_inc_P_e
	Function3	gsl_sf_gamma_inc_e
hyperg_1F1.c	**Function1**	hyperg_1F1_1_small_a_bgt0
	Function2	hyperg_1F1_ab_posint
	Function3	hyperg_1F1_ab_pos
	Function4	hyperg_1F1_ab_neg
	Function5	gsl_sf_hyperg_1F1_inc_e

early in training because it can easily distinguish between real data and generated data. Once the Generator is sufficiently trained, the Discriminator will have difficulty distinguishing between real and generated data. This causes the Discriminator to constantly make errors and obtain large gradients.

We have considered two modules in the GNU Scientific Library (a numerical library for C and C++ programmers) [23]: the `gamma_inc.c` module and the `hyperg_1F1.c` module. The `gamma_inc.c` module is used to compute the incomplete Gamma function. It contains 13 functions including branches, where the total number of branches is 138. The `hyperg_1F1.c` module is used to calculate the confluent hypergeometric function. It contains 21 functions including branches, where the total number of branches is 592. In the integration testing, the GAN model generates path information including all branches. In the unit testing, we focus on several functions in these modules. Concretely, 3 functions in `gamma_inc.c` module and 5 functions in `hyperg_1F1.c` module are picked up to make their test coverage increase (see Table 4.6).

We compared the test coverage of the test inputs generated by the experiment and compared the accuracy of the generated path, to verify whether the framework can generate test inputs that can improve the test coverage, and to investigate which version of GAN model is more suitable for our framework. We also conducted a random test to generate 200 test inputs to compare the test coverage with the GAN method.

2.4.2 Results

Tables 3.3 to 3.4 show the results of integration testing and unit testing under three models and the random test's test coverage. The initial coverage represents the percent of branches executed by 100 initial training data. The trained coverage represents the percentage of branches executed by 200 input data, where 200 input data contains 100 initial training data and 100 GAN generated input data. The accuracy is calculated by Eq. (1). In order to facilitate comparison, we divide the data generated in the integration testing into function units to compare with the unit testing results. Figs. 2.7 to 2.10 present the analysis of results. As can be seen in Fig. 2.7 that in addition to Function 1 in the hyperg_1F1.c module, the test coverage of input data generated by GAN (dense) in unit tests is higher than that of input data generated in integration tests. Also, the test coverage of input data generated by GAN (conv) in unit tests is higher than that of input data generated in integration tests except for Function 2 and Function 4 in the hyperg_1F1.c module. Despite of this result, only a few of the input data generated by the GAN model have a coverage rate that exceeds the random testing. Furthermore, BiGAN does not show good performance in our framework, because the coverage of the input data generated by BiGAN is also mostly not better than random testing (see Fig. 2.8). However, in the unit testing, the coverage of test inputs generated by WGAN-GP (dense) is higher than that of randomly generated test inputs except for Function 1 and Function 5 in the hyperg_1F1.c module. Further, the WGAN-GP (conv) improves the test coverage of Function 3 in hyperg_1F1.c module and Function 2 in gamma_inc.c module. Fig. 2.10 graphically shows the path prediction accuracy of three sets of models. On the whole, the WGAN-GP (dense), the GAN (dense), and the GAN (conv) have higher path prediction accuracy than the others in unit testing. To this end, we conclude that our framework is more appropriate for unit testing. This is because taking the function as a unit can reduce the complexity of the program and improve the model's path prediction accuracy.

$$\text{Accuracy} = \frac{\text{The number of correct branchs}}{\text{The number of total branchs}} \times 100\% \tag{2.1}$$

We also analyze the distribution of input data generated by three types of GAN models. Fig. 2.11- Fig. 2.16 illustrate the input data distribution of 5

Table 2.2: GAN: Comparison of the coverage achievement and the accuracy of path.

		No. of branches	**methods**	**GAN(dense)**		**GAN(conv)**		**RAND**
				Initial coverage	**Trained coverage**	**Initial coverage**	**Trained coverage**	**Random coverage**
hyperg_1F1.c	**All**	**594**	**Integration**	34.84%	42.93%	34.84%%	51.34%	45.12%
	Function1	**38**	**Integration**	47.36%	60.52%	47.36%	52.63%	63.15%
			Unit	36.84%	55.26%	36.84%	57.89%	
	Function2	**70**	**Integration**	41.42%	52.85%	41.42%	72.85%	72.85%
			Unit	41.43%	65.71%	41.43%	64.28%	
	Function3	**96**	**Integration**	35.41%	41.66%	35.41%	65.62%	76.04%
			Unit	71.87%	77.08%	71.87%	79.16%	
	Function4	**58**	**Integration**	44.82%	67.24%	44.82%	79.31%	77.58%
			Unit	56.89%	70.68%	56.89%	74.13%	
	Function5	**42**	**Integration**	63.15%	78.94%	63.15%	78.57%	84.21%
			Unit	59.52%	80.95%	59.52%	80.95%	
gamma_inc.c	**All**	**138**	**Integration**	49.27%	51.45%	49.27%	52.17%	50.00%
	Function1	**24**	**Integration**	45.83%	45.83%	45.83%	54.16%	66.66%
			Unit	54.16%	70.83%	54.16%	58.33%	
	Function2	**20**	**Integration**	55.00%	70.00%	55.00%	65.00%	75.00%
			Unit	75.00%	75.00%	75.00%	75.00%	
	Function3	**16**	**Integration**	33.33%	33.33%	33.33%	33.33%	27%
			Unit	68.75%	68.75%	68.75%	75.00%	

Table 2.3: BiGAN: Comparison of the coverage achievement and the accuracy of path.

		No. of branches	**methods**	**BiGAN(dense)**		**BiGAN(conv)**		**RAND**
				Initial coverage	**Trained coverage**	**Initial coverage**	**Trained coverage**	**Random coverage**
hyperg_1F1.c	**All**	**594**	**Integration**	34.84%	34.65%	34.84%	46.80%	45.12%
	Function1	**38**	**Integration**	47.36%	47.36%	47.36%	52.63%	63.15%
			Unit	36.84%	52.63%	36.84%	52.63%	
	Function2	**70**	**Integration**	41.42%	41.42%	41.42%	67.14%	72.85%
			Unit	41.43%	41.43%	41.43%	65.71%	
	Function3	**96**	**Integration**	35.41%	43.75%	35.41%	55.20%	76.04%
			Unit	71.87%	72.91%	71.87%	76.04%	
	Function4	**58**	**Integration**	44.82%	40.74%	44.82%	65.51%	77.58%
			Unit	56.89%	60.34%	56.89%	62.06%	
	Function5	**42**	**Integration**	63.15%	73.68%	63.15%	78.57%	84.21%
			Unit	59.52%	71.42%	59.52%	76.19%	
gamma_inc.c	**All**	**138**	**Integration**	49.27%	56.52%	49.27%	52.89%	50.00%
	Function1	**24**	**Integration**	45.83%	66.66%	45.83%	54.16%	66.66%
			Unit	54.16%	75.00%	54.16%	58.33%	
	Function2	**20**	**Integration**	55.00%	80.00%	55.00%	70.00%	75.00%
			Unit	75.00%	85.00%	75.00%	75.00%	
	Function3	**16**	**Integration**	33.33%	33.33%	33.33%	33.33%	27%
			Unit	68.75%	68.75%	68.75%	75.00%	

Table 2.4: WGAN-GP: Comparison of the coverage achievement and the accuracy of path.

		No. of branches	methods	WGAN-GP(dense)		WGAN-GP(conv)		RAND
				Initial coverage	Trained coverage	Initial coverage	Trained coverage	Random coverage
hyperg_1F1.c	All	594	Integration	34.84%	51.42%	34.84%	51.34%	45.12%
	Function1	38	Integration	47.36%	57.89%	47.36%	60.52%	63.15%
			Unit	36.84%	63.15%	36.84%	55.26%	
	Function2	70	Integration	41.42%	62.85%	41.42%	74.28%	72.85%
			Unit	41.43%	84.28%	41.43%	75.71%	
	Function3	96	Integration	35.41%	75.00%	35.41%	64.58%	76.04%
			Unit	71.87%	83.33%	71.87%	90.62%	
	Function4	58	Integration	44.82%	74.13%	44.82%	72.41%	77.58%
			Unit	56.89%	84.48%	56.89%	70.68%	
	Function5	42	Integration	63.15%	76.31%	63.15%	76.19%	84.21%
			Unit	59.52%	80.95%	59.52%	76.19%	
gamma_inc.c	All	138	Integration	49.27%	52.17%	49.27%	52.89%	50.00%
	Function1	24	Integration	45.83%	54.16%	45.83%	54.16%	66.66%
			Unit	54.16%	75.00%	54.16%	66.66%	
	Function2	20	Integration	55.00%	65.00%	55.00%	70.00%	75.00%
			Unit	75.00%	80.00%	75.00%	90.00%	
	Function3	16	Integration	33.33%	33.33%	33.33%	45.45%	27%
			Unit	68.75%	80.00%	68.75%	75.00%	

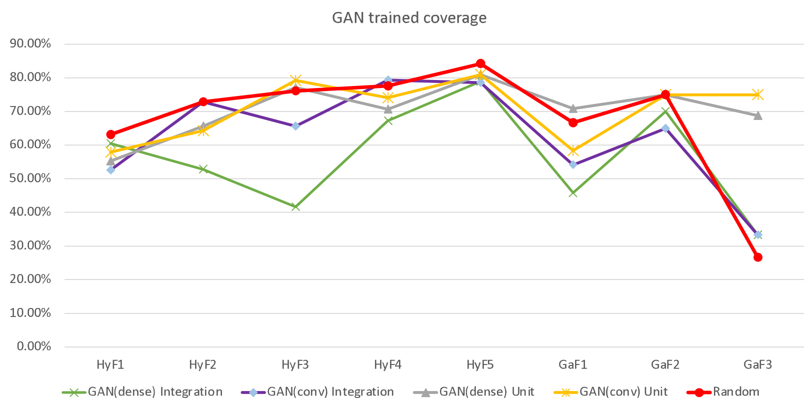


Figure 2.7: Coverage of GAN generated input data.

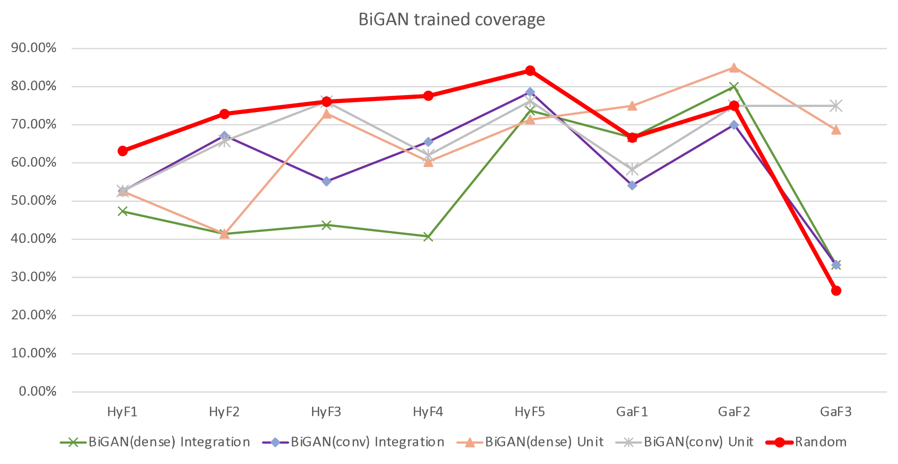


Figure 2.8: Coverage of BiGAN generated input data.

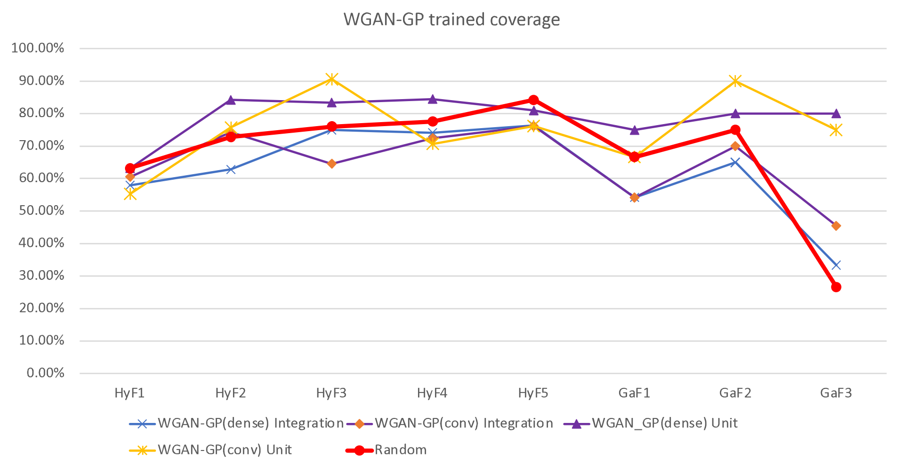


Figure 2.9: Coverage of WGAN-GP generated input data.

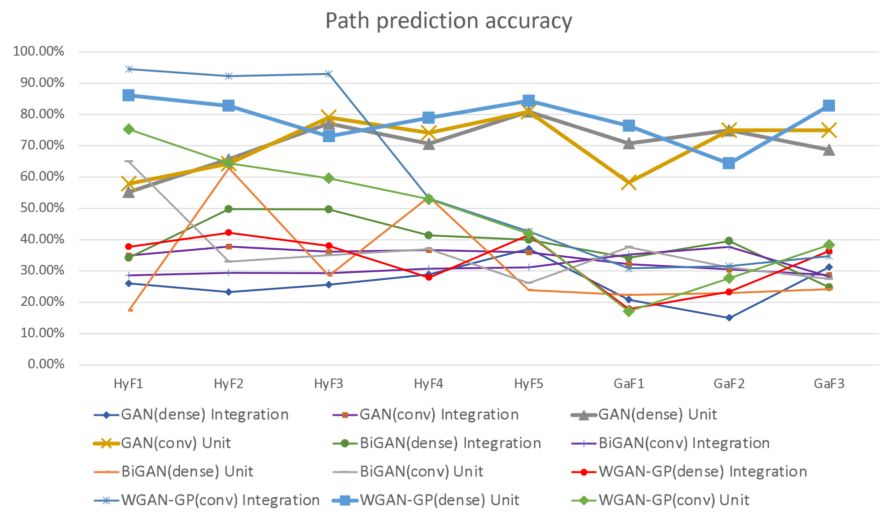


Figure 2.10: Path prediction accuracy.

functions in hyperg_1F1.c module generated by the standard GAN, biGAN, and WGAN-GP, respectively. As can be seen from these figure, both fully connected BiGAN and convolutional BiGAN have problems with mode collapse, that is, the generator collapses and produces a limited variety of samples. Meanwhile, the input data generated by the GAN model using the fully connected neural network is also easy to move to the edge. Only convolutional GAN and WGAN-GP can generate diverse input data for our strategy. In summary, compared with standard GAN and BiGAN, WGAN-GP can avoid the mode collapse problem and create new input data instances that resemble our training data, so that it showed the better performance in our framework.

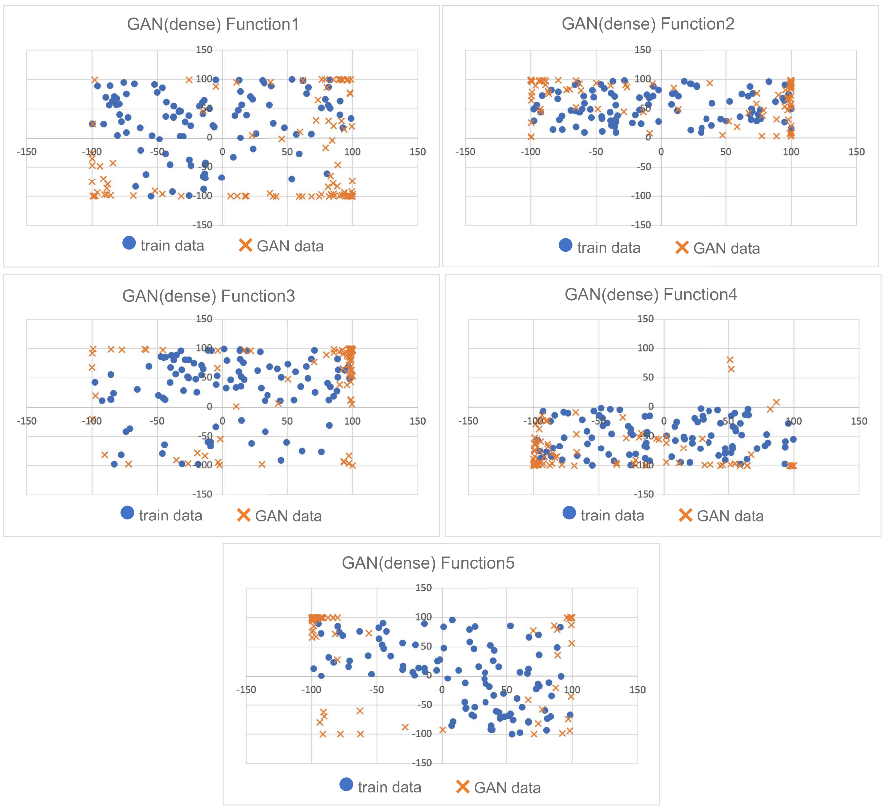


Figure 2.11: Data distribution is generated by the standard GAN with fully connected network.

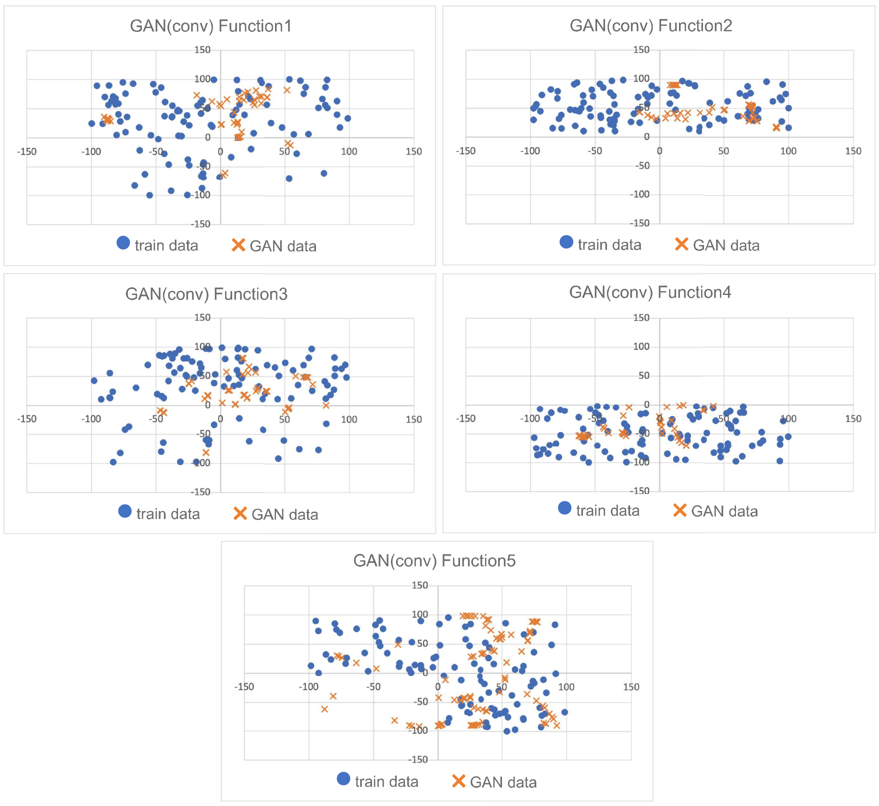


Figure 2.12: Input data distribution is generated by the standard GAN with convolutional network.

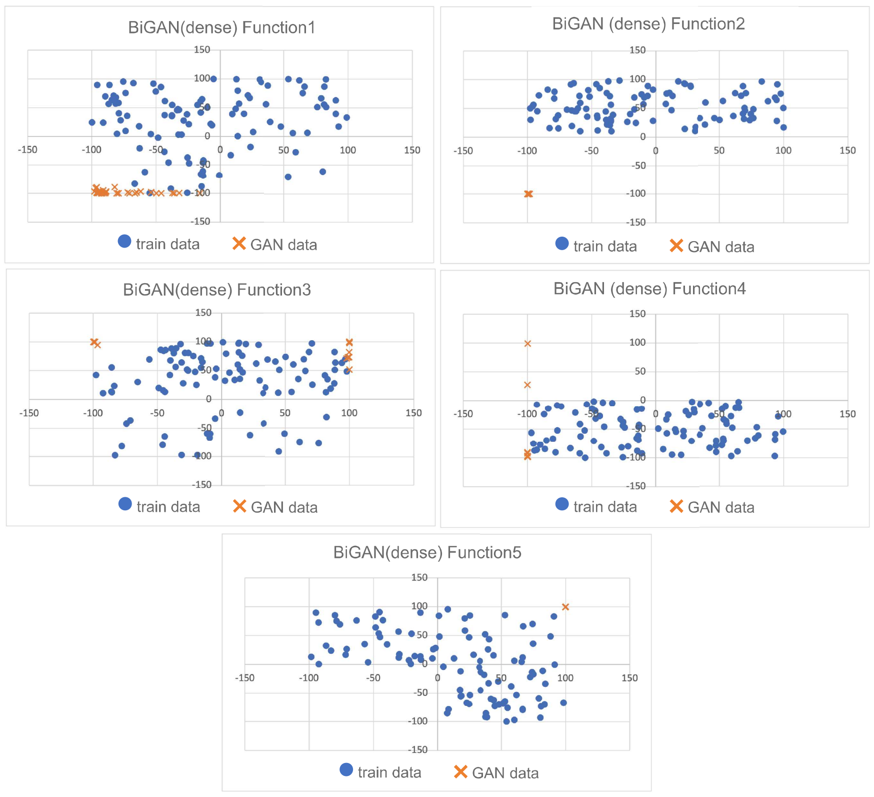


Figure 2.13: Input data distribution is generated by BiGAN with fully connected network.

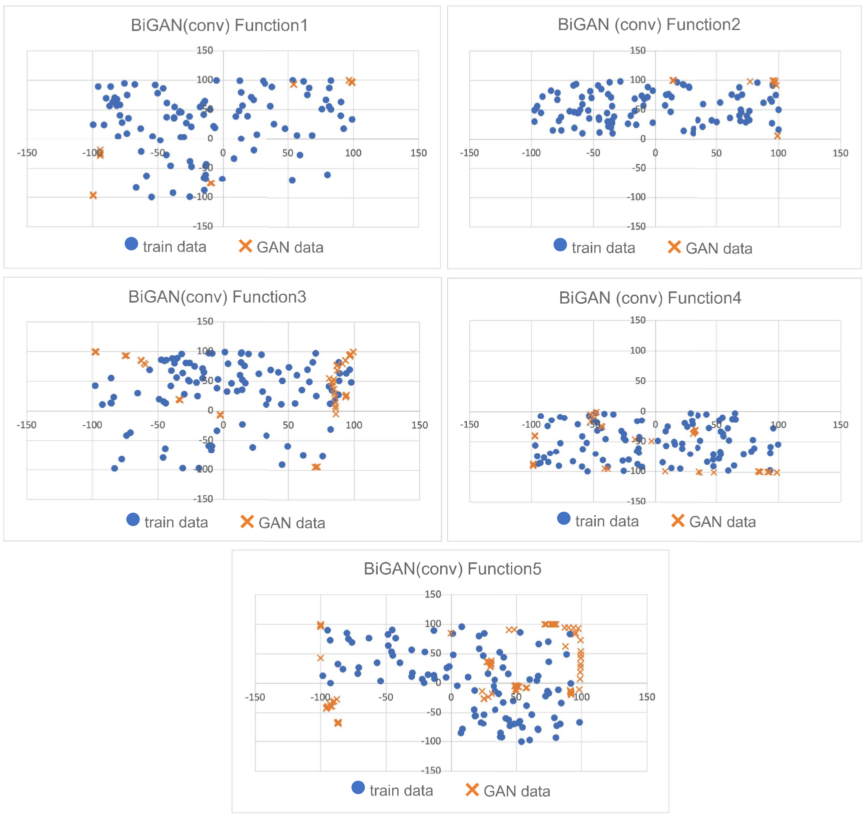


Figure 2.14: Input data distribution is generated by BiGAN with convolutional network.

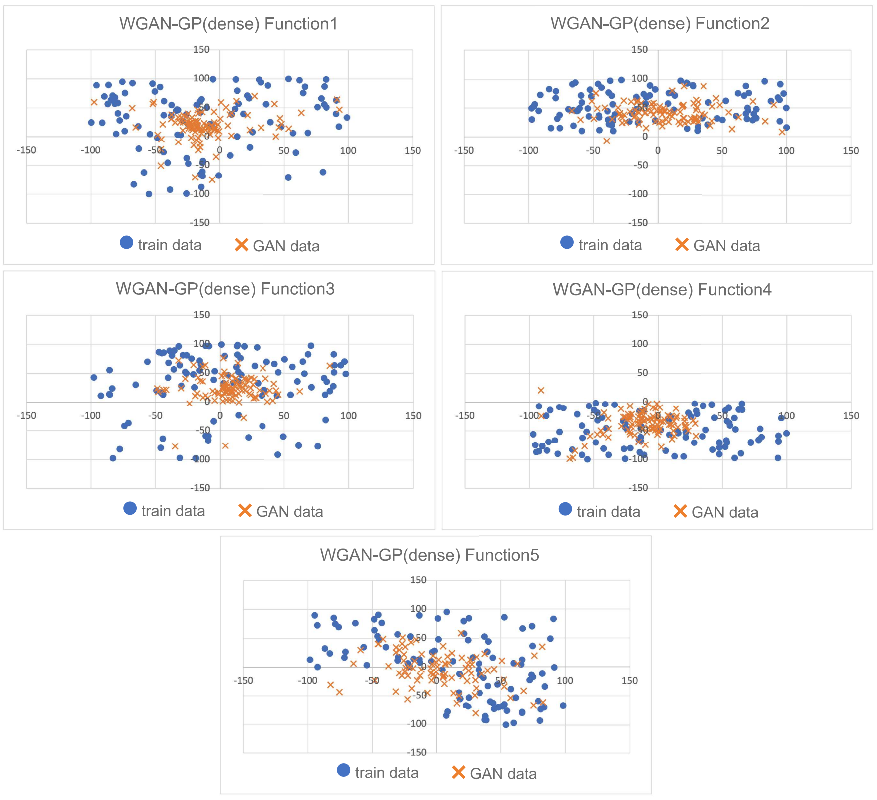


Figure 2.15: Input data distribution is generated by WGAN-GP with fully connected network.

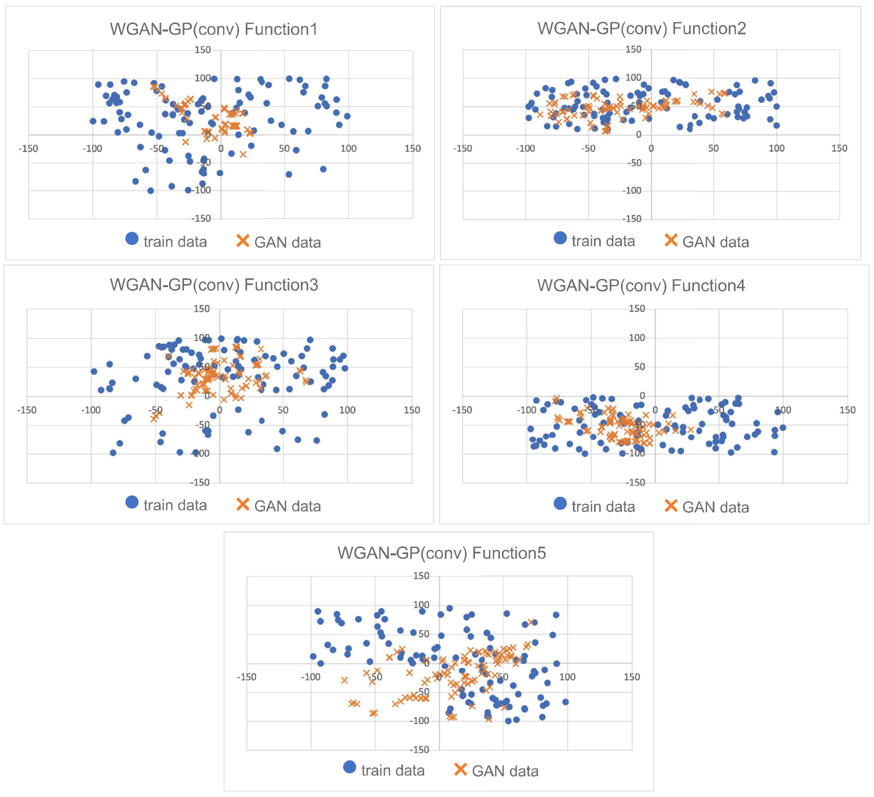


Figure 2.16: Input data distribution is generated by WGAN-GP with convolutional network.

Chapter 3

Test Input Generation with MLP-Based Boundary Value Analysis

3.1 Introduction

In this chapter, we propose an MLP (Multilayer Perceptron) based approach for the BVA in white-box testing. The idea behind our approach is twofold. First we train an MLP-based discriminator that answers whether two test inputs have the same execution path or not. Second we generate test inputs based on Markov Chain Monte Carlo (MCMC) with the outputs of discriminator. Zhou et al. [24] first utilized MCMC methods for software random testing. MCMC-RT (MCMC Random Testing) uses the prior knowledge of software testing and is based on the statistical background of the probability of failure described by Bayesian inference. Our idea is an extension of MCMC-RT by introducing the MLP-based discriminator. Our method needs the information on execution paths only, and does not require any program information identifying conditions.

3.2 Generation of Boundary Values

3.2.1 Markov Chain Monte Carlo (MCMC)

MCMC is a general technique for efficiently generating samples drawn from a probability distribution with high-dimensional space. The idea of the MCMC is to simulate an ergodic Markov chain whose stationary distribution is consistent with a target distribution. The more steps of Markov chain simulation, the more closely the distribution of the sample matches the actual desired distribution.

It is significant to construct an appropriate Markov chain when using the MCMC method to generate samples. Different transfer construction methods will produce different MCMC methods. At present, the commonly used MCMC methods mainly include two Gibbs sampling [25] and Metropolis-Hastings (M-H) algorithm [26]. Since Gibbs sampling is a special case of the M–H algorithm. Here we only summarize the M-H algorithm.

The M-H algorithm produces a Markov chain whose limiting distribution is the target density $\pi(x)$. Let x' be a candidate of the next state of Markov chain that is generated from a proposal distribution $Q(x'|x)$ where x is the current state of Markov chain. This candidate becomes the next state of the Markov chain with the following acceptance probability:

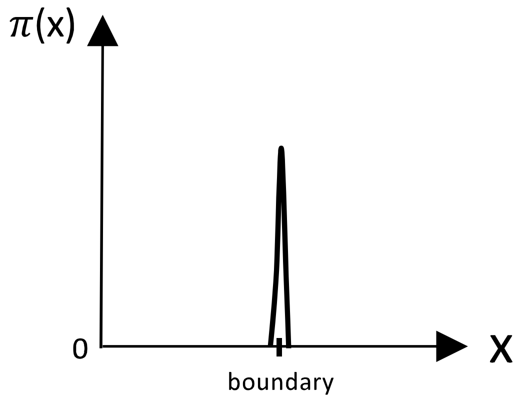
$$P = \min\left(\frac{\pi(x')Q(x|x')}{\pi(x)Q(x'|x)}, 1\right) \tag{3.1}$$

In practice, we generate a uniform random number U, if U is less than or equal to the acceptance probability $U \leq P$, the candidate is accepted, otherwise, the candidate is rejected. After repeating this process for several steps, the sample x can be regarded as a sample drawn from $\pi(x)$.

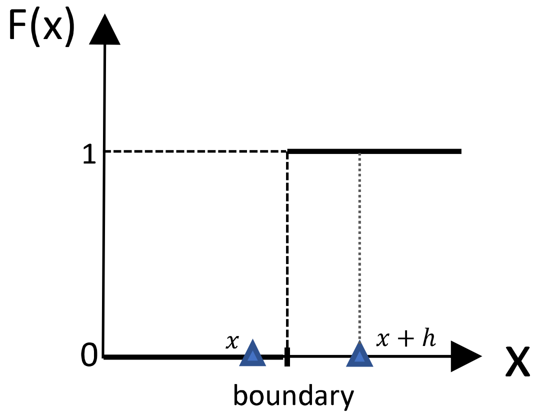
3.2.2 MCMC for Boundary Values

Consider the MCMC for generating boundary values. As mentioned before, MCMC is essentially a method to generate samples from the target density $\pi(x)$. Our idea is to estimate the density function $\pi(x)$ on the input domain, which represents the likelihood that whether x is a boundary value or not. When $\pi(x)$ is used as the target density of MCMC, the samples generated by MCMC are expected to be boundary values.

The key issue is how to estimate such a density function for boundary value. As an example, we consider a program that has only one input value x. If the program has one boundary, the density function for boundary value is expressed as a function with rapidly climbing and falling like in Fig. 3.1a. In the sense of mathematics, it is a delta function, and it is not easy to estimate such a function directly. On the other hand, Fig. 3.1b shows the cumulative distribution of Fig. 3.1a. Although it is a step function in the mathematical sense, it is possible to approximate such a function by a continuous function like a logistic function. If



(a)



(b)

Figure 3.1: The probability density and the cumulative distribution of boundary value.

we obtain the cumulative distribution for boundary value $F(x)$, then the target density is approximated as

$$\pi(x) = \frac{F(x+h) - F(x)}{h}, \tag{3.2}$$

where h is an arbitrary and sufficiently small value. Also, since $F(x)$ jumps at around the boundary, the value $F(x+h) - F(x)$ takes 1 when the test inputs $x+h$ and x belong to different equivalence partitions. Otherwise, if $x+h$ and x are in the same equivalence partition, $F(x+h) - F(x)$ becomes 0.

This idea is expanded to the case where high-dimensional test input space. Let $N(x, y)$ be the function meaning that $N(x, y) = 1$ if inputs x and y are in the different equivalence partitions. Otherwise, if inputs x and y are in the same equivalence partition, $N(x, y) = 0$. Then the target density is given by

$$\pi(x) = \frac{1}{m} \sum_{i=1}^{m} \frac{N(x+h_i, x)}{h_i}, \tag{3.3}$$

where h_i is a small vector to put a perturbation. Figure 3.2 illustrates our approach in two-dimensional space. In the figure, there are two inputs a and b and one boundary. The circles represent the radius R of perturbation. Since X' is closer to the boundary than X, the likelihood that X' and $X' + h_i$ belong to the different equivalence partitions is higher than X. That is, $\pi(X')$ may be greater than $\pi(X)$.

Based on the density function for boundary value $\pi(x)$, the acceptance probability in the M-H algorithm becomes

$$P = \min\left(\frac{\sum_{i=1}^{m} \frac{N(x'+h_i, x')}{h_i} Q(x|x')}{\sum_{i=1}^{m} \frac{N(x+h_i, x)}{h_i} Q(x'|x)}, 1 \right) \tag{3.4}$$

One of the purposes of generating boundary values is to generate test inputs for software testing. In this sense, it is better that the generated boundary values cover the input domain of the software. In the M-H algorithm, the proposal distribution $Q(x'|x)$ is frequently designed by searching inputs close to the original (current) input x, i.e., the local search. However, the local search cannot ensure the coverage of test domain, and thus the paper considers the independent proposal distribution that does not depend on the original (current) input $Q(x')$. The typical example of such proposal distribution is the uniform distribution on input domain. When we use the uniform distribution on input

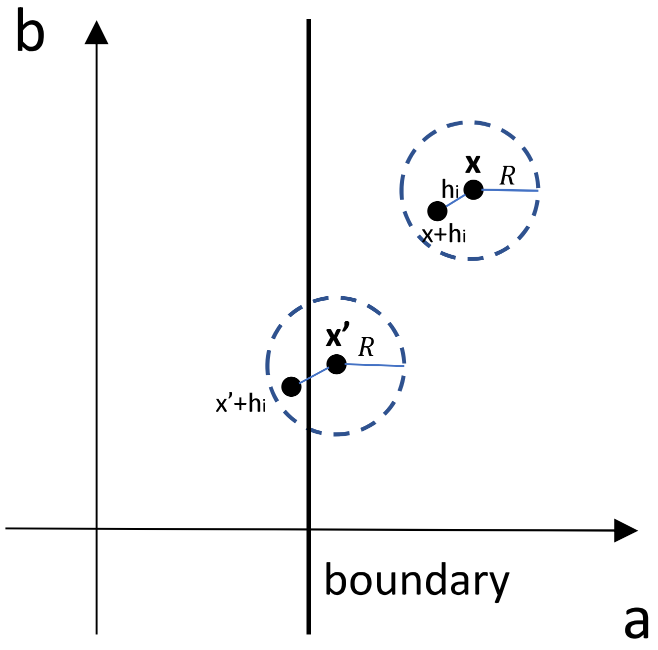


Figure 3.2: An example in two-dimensional space.

domain, the acceptance probability simply becomes

$$P = \min\left(\frac{\sum_{i=1}^{m}\frac{N(x'+h_i,x')}{h_i}}{\sum_{i=1}^{m}\frac{N(x+h_i,x)}{h_i}}, 1\right). \tag{3.5}$$

3.3 Exploitation of MLP Model

3.3.1 Model Architecture

To generate boundary values, we need the function $N(x, y)$ that outputs whether two inputs x and y belong to the different equivalence partitions. The simplest and direct approach is to monitor concrete paths by executing software with two inputs. However, since we need a number of executions of $N(x, y)$ in the scheme of MCMC, the direct approach is not appropriate in this case. The second way is to create the function $N(x, y)$ with the static analysis such as symbolic execution beforehand. This method may be effective but the static analysis has a limitation on scalability. In this chapter, we exploit a MLP model to create the function $N(x, y)$. The MLP-based approach is one of the data-driven approaches. The MLP model is trained from the data, and the model mimics the trained data and interpolates unknown two inputs predicatively. Although the training of model requires much computational cost, the evaluation is done with less computation cost. This property is appropriate for the function in MCMC scheme.

In this chapter, we use a MLP to represent the function $N(x, y)$. MLP are multilayer perceptrons, including an input layer or multiple hidden layers and an output layer. All these units are connected to each other through links with synaptic weights. These weights are updated as part of the training process and reflect the information learned during the training process. In our method, the input of the MLP is a set of input pairs, such as the vector $(x, x + h)$ and the output of the MLP is a label as a binary value indicating whether x and y are in the same partition or not.

3.3.2 Training Data

Before using MLP prediction, we first need to train the MLP with a set of training data. Since we focus on the white-box boundary value where equivalence partitions are defined by execution paths of program for test inputs, it is necessary to define the equivalence of execution path based on the execution log. In addition, MLP requires a number of training data, and thus the training data collection should be automatically executed. In this chapter, we provide the approach based on Gcov tool.

In white-box boundary value testing, we want to cover the boundary values for comparison predicates. Each atomic Boolean expression in the path condition is referred to here as a predicate. Predicates could be Boolean variables, comparison predicates ($>, >=, <, <=, =, \neq$), etc., and should not contain any Boolean operator (such as $\wedge, \vee, \neg$, etc.) [16]. Each comparison predicate contains two branches, and each branch has three states, marked as "`1, 0, -1`". Suppose we have a comparison predicate ($a > 0$), contains two branches: ($a > 0$) and ($a <= 0$). We use Gcov to extract the execution path when testing the program, as described in Chapter 2.2. Then the label of input pairs is obtained by calculating whether the execution paths corresponding to the two inputs are equal. If they are equal, the label is 0 (there is no boundary between the two inputs), and if they are not equal, the label is 1 (there is a boundary between the two inputs). In this chapter, we apply the MLP-based approach to the c language. For other languages, we can apply this method by simply changing the way paths are extracted.

3.4 Experiment

In this section, we present experiments to investigate the effectiveness of MLP-based approach. We conducted two sets of experiments. One is to generate test inputs for a simple C program and compare the effects of various parameter combinations. Another one is an experiment on several real programs.

3.4.1 Fault detection ability

We use mutation testing to study the fault detection rate of test sets. Mutation testing is a type of software testing in which certain statements of the source code are changed/mutated to check if the test inputs are able to find errors in the source code. In the experiment, we injected (seeded) several faults into the program. Each seeded fault yields a faulty version. For each test set generated in the experiments, we run the whole test set on each faulty version and count the number of killed mutations. The kill rate is calculated by Eq. (3.6).

$$kill\_rate = \frac{the\ number\ of\ killed\ mutations}{total\ number\ of\ mutations} \tag{3.6}$$

We manually inject (seeded) 6 kinds of faults in the program under test. Off-by-one bugs (OBOB) are a kind of faults when some computation process uses some wrong value which is 1 more or 1 less than the correct value, and most boundary faults are Off-by-one bugs [16]. The rest of faults contain five common mutation operators [27]: constant replacement (CR), relational operator replacement (ROR), arithmetic operator replacement (AOR), scalar variable replacement (SVR), and logical operator replacement (LOR). We use the execution path and the output to determine if a fault is killed, that is, when at least one test input has an execution path different from the correct version, or at least one test input has an output different from the correct version, the mutation is killed.

3.4.2 RT and ART

In the RT approach, we randomly generate a test set consisting of n test inputs and execute each mutated program with this test set to study the fault detection ability.

ART is executed by the algorithm described in [28]. In traditional ART algorithm, the executed set is incrementally updated with the selected element from the candidate set until a failure is revealed. However, to make the experimental results of ART and MLPBVA comparable, we changed the experimental stopping condition of ART to generate n test inputs incrementally. For each mutation, the test input generation process stops if the injected fault is detected when generating the i-th ($i \leq n$) test input, and the generated test set kills the mutation. If none of the generated n test inputs detect the fault, the generated test set did not kill the mutation.

3.4.3 Experiment with a Simple C Program

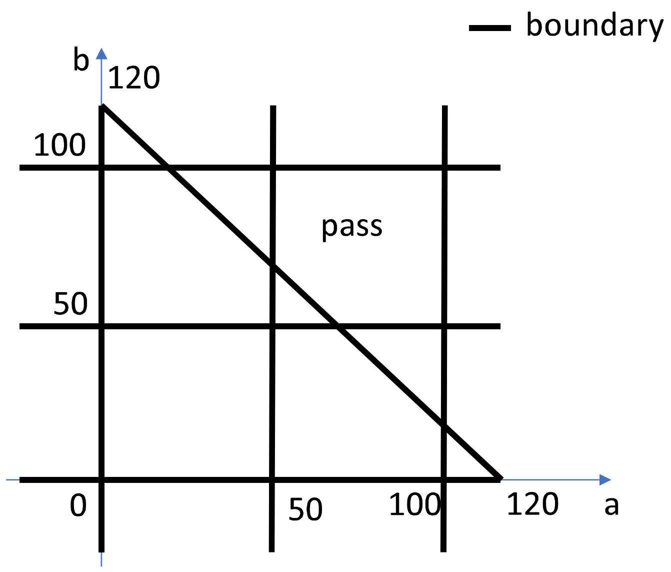


Figure 3.3: Conceptual diagram of the boundaries.

In this experiment, we use a program `En_testing.c` described in the Chapter 2.2. Figure 3.3 is a conceptual diagram of the boundaries. We injected (seeded) 25 faults into the program. Among them, 14 faulty versions contain off-by-one bugs (where we add/subtract 1 to the right-hand side of comparison predicates), and the rest of the faults contain five common mutation operators.

Design Parameters of MLP and MCMC: The MLP in our experiments is a fully connected NN (dense layers) with one input layer, two hidden layers, and one output layer. Each hidden layer has 64 units and the activation function is 'relu'. The input size of the input layer is 4. The output layer has an output size of 1 and the activation function is 'sigmoid'. In addition, the parameter learning rate and the number of training epochs (one epoch is the cycle when an entire dataset is passed forward and backward through the neural network only once.) are set to 0.01 and 50, respectively. In our experiments, we use three sets of initial training data, which are randomly generated from the input domain. The size of the dataset is 20000, 10000, and 5000, respectively, which are marked as Dataset20000, Dataset10000, and Dataset5000.

Let $f(x) = \sum_{i=1}^{m} N(x + h_i, x)$, When both $f(x')$ and $f(x)$ are calculated as 0, we cannot judge which of the current sample x and the candidate sample x' is closer to the boundary. So in the experiment, we directly use the probability value output by the neural network as the value of f(x).

For MCMC, we use the uniform distribution as the proposal distribution. We use MCMC model to generate n test inputs from input domain and examine the cases $n = 5$, $n = 10$, $n = 20$, and $n = 50$. Selecting a point close to the boundary as the initial value of MCMC is more conducive to generating the boundary value. Therefore, we choose $(49, 49)$ as the initial value of MCMC. In our experiments, MCMC samples at each step and stops until n test inputs are generated. And the MLP will be retrained for every 10 steps.

Results and Discussion: We use RT, ART, and MLP-based approach to generate test inputs for a simple program under test. Figure 3.4 shows the input data distribution generated by RT. Figure 3.5 shows the input data distributions generated by MLP-based approach under parameters {n=5, n=10, n=20, n=50} with Dataset20000, Dataset10000, and Dataset5000, respectively. It can be seen intuitively from the distribution graph that the MLP-based approach can generate test inputs near the boundary. And to cover all boundaries, we need more test inputs, such as n=50.

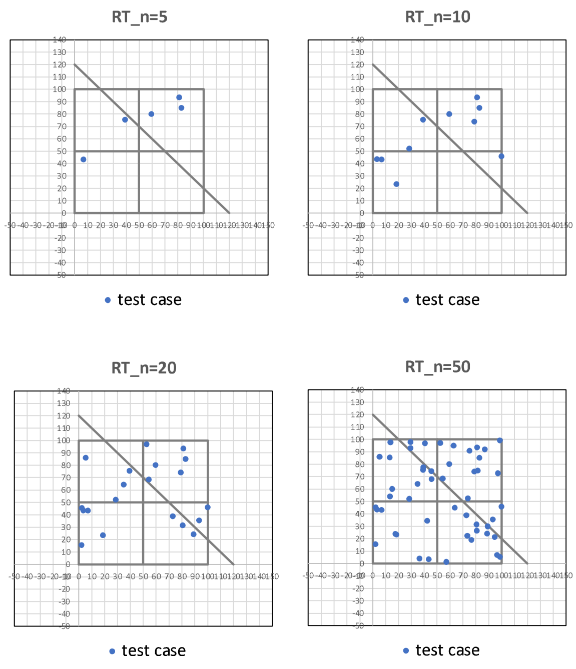


Figure 3.4: The input data distribution generated by RT.

Table 3.1: Kill rate of RT, ART and MLP-based boundary value analysis (MLP-BVA).

method	**n=5**	**n=10**	**n=20**	**n=50**
RT	0.28	0.36	0.36	0.4
ART	0.28	0.28	0.4	0.44
MLPBVA (Dataset20000)	0.36	0.36	0.56	0.68
MLPBVA (Dataset10000)	0.28	0.4	0.52	0.64
MLPBVA (Dataset5000)	0.4	0.4	0.48	0.64

We also record the number of faults detected by each test set. Table 3.1 shows the kill rates corresponding to the test sets generated by various methods. The results show that the kill rate of most test sets generated by MLP-based approach is better than that of RT and ART. Compared with RT and ART, our proposed MLP-based approach can generate better quality test inputs and detect more faults.

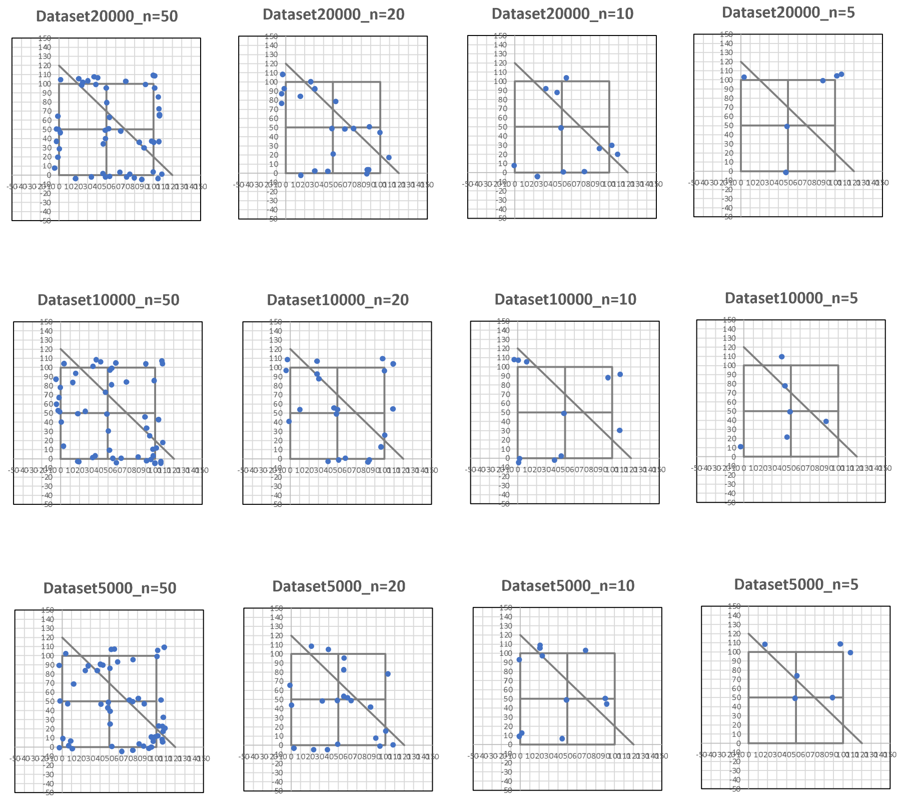


Figure 3.5: The input data distribution generated by MLP-based approach with Dataset.

3.4.4 Experiment with Real Programs

We select seven programs used in the existing literature to evaluate the effect of our approach on real programs. Machine learning can handle both numerical data and categorical data. During the data processing phase, machine learning models convert structured data into numerical data because the input layer of the neural network only accepts numerical input data. Therefore, in this experiment, we select seven programs with inputs of only numeric types, containing continuous data and discrete data. The descriptions of these subject programs are shown in Table 4.6. And the details about all seven programs are shown in Table 3.3, such as the dimensional number of program inputs, the range of input domain, line of code (LOC), fault information, and the number of boundary

Table 3.2: Experimental programs

Prog Name	Description
triType [29]	The type of a triangle
nextDate [30]	Calculate the following date of the given day
findMiddle [31]	Find the middle number among three numbers
bessj [32]	Bessel function J of general integer order
expint [32]	Exponential integral
plgndr [32]	Legendre polynomials
tcas [33]	Aircraft collision avoidance system

Table 3.3: Details of 7 subject programs

Program	Dim	Input Domain		Size(LOC)	Fault types						Total Faults	num_Bvalues
		From	To		OBOB	CR	ROR	AOR	SVR	LOR		
triType	3	(0, 0, 0)	(100, 100, 100)	41	2		7	2	1	6	18	15
nextDate	3	(1800, -2, -2)	(3000, 14, 33)	90	16		7			8	31	42
findMiddle	3	(-100, -100, -100)	(100, 100, 100)	36			14			5	19	15
bessj	2	(2,-1000)	(300, 15000)	133	10	1	11	1	4		27	21
expint	2	(-10 , -10)	(1500, 1500)	109	12	2	2	5	3	10	34	6
plgndr	3	(-5,-5,-5)	(5,5,5)	65	6		6	2	3	2	19	12
tcas	12	(0,0,0,0,0,0, 0,0,0,0,0,0)	(1000,1,1,50000,1000,50000, 3,1000,1000,2,2,1)	182	17	4	2		5	6	34	19

values.

Concolic testing: Concolic testing is a hybrid software verification technique that performs symbolic execution along a concrete execution path. Symbolic execution is a software testing technique that substitutes symbolic values for normal inputs to a program during program execution. By symbolizing the program inputs, the symbolic execution maintains a set of constraints for each execution path. After the execution, constraint solvers will be used to solve the constraint and determine what inputs cause the execution. Its purpose is to maximize code coverage. KLEE is a dynamic symbolic execution tool built on the LLVM compilation framework that automatically generates test inputs and achieves high program coverage [34]. In this experiment, we use KLEE to generate test inputs for seven programs and compare the kill rate with our proposed MLP-based method.

Manually-Performed Boundary Value Analysis:

In this experiment, we asked a student to generate a set of boundary values by manually analyzing the source code for comparing the fault detection ability with the MLP-based approach.

In this work, the student uses an input that is on the edge of a given predicate as a boundary value. Suppose there is a path condition $(a > 0) \vee (b <= 0)$, including two boundaries $a = 0, b = 0$, therefore, the input (a,b)=(0,0), input (a,b)=(0,1) and input (a,b)=(1,0) can be used as three boundary values. In our experiment, the student selects the endpoints of all boundary lines in the input domain and the intersection points between the boundary lines as boundary values. We will compare the fault detection ability of boundary values obtained by manually analyzing the source code with the fault detection ability of MLP-based approach that do not manually analyze the source code. The number of selected boundary values (num_Bvalues) for 7 programs are shown in Table 3.3.

Results and Discussion:

We examine the fault detection capabilities of RT, ART, Manually-performed boundary value analysis approach, concolic testing and MLP-based approach, respectively. Because our methods show better performance when the number of initial datasets is n=50 in the experiment with a simple program. In the real program experiment, the MCMC generates $n = 50$ test inputs from the input domain.

Table 3.4 shows the comparison of kill rate with RT, ART, Manually-performed boundary value analysis approach (BVA), Concolic testing (KLEE), and MLP-based approach (MLPBVA). KLEE generates n test inputs for each of the seven programs. Without analyzing the source code, our proposed MLP-based approach outperforms the Manually-performed boundary value analysis approach in four of the seven programs tested. And MLP-based approach has better fault detection ability than RT and ART in program tests except for `tcas` program. Meanwhile, compared with the concolic testing method, the kill rate of test inputs generated by the MLP-based method outperforms concolic testing among

Table 3.4: Comparison of kill rate with RT, ART, Manually-performed boundary value analysis approach (BVA), Concolic testing (KLEE), and MLP-based approach (MLPBVA)

Method	Kill rate						
	triType	nextDate	findMiddle	bessj	expint	plgndr	tcas
RT	0.61	0.58	0.36	0.52	0.47	0.63	0.29
ART	0.61	0.51	0.63	0.66	0.47	0.58	0.05
BVA	0.88	0.45	**1**	0.7	0.7	0.78	0.06
MLPLBVA (Dataset20000)	0.72	0.65	**1**	0.81	0.68	**0.95**	0.05
MLPBVA (Dataset10000)	0.72	0.7	0.79	**0.85**	**0.79**	**0.95**	0.05
MLPBVA (Dataset5000)	0.66	0.68	0.89	0.59	0.74	0.84	0.05
KLEE	**1 (n=14)**	**0.9 (n=56)**	**1 (n=13)**	0.37 (n=2)	0.38 (n=4)	0.84 (n=14221)	**1 (n=1290)**

Table 3.5: Prediction accuracy

Initial_trainData	MLP prediction accuracy						
	triType	nextDate	findMiddle	bessj	expint	plgndr	tcas
Dataset20000	0.53	0.54	0.84	0.75	0.48	0.84	0.64
Dataset10000	0.54	0.65	0.78	0.61	0.44	0.81	0.69
Dataset5000	0.46	0.64	0.37	0.33	0.39	0.77	0.78

Table 3.6: Time cost

method	Time (sec)						
	triType	nextDate	findMiddle	bessj	expint	plgndr	tcas
RT	249	452	287	384	514	259	449
ART	248	572	193	320	190	523	825
MLPBVA (Dataset20000)	11605	11762	11950	11970	11828	11330	11679
MLPBVA (Dataset10000)	6060	6292	6273	6221	6291	5871	6153
MLPBVA (Dataset5000)	3319	3400	3242	3403	3464	3169	3392
KLEE	69 (n=14)	470 (n=56)	68 (n=13)	21 (n=2)	43 (n=4)	76244 (n=14221)	11650 (n=1290)

the three programs.

Whether the MLP-based approach can generate high-quality test inputs near the boundary highly depends on the accuracy of the neural network's prediction of whether the sample is near the boundary. Table 4.5 shows the prediction accuracy of the neural network in the one-step sampling process of MCMC. Overall, when the neural network has higher prediction accuracy, the fault detection ability of the generated test inputs will be stronger. In order to use neural networks to generate higher-quality test inputs, we need to consider improving the performance of neural networks in the future, so that neural networks can better learn the boundary information of programs.

Table 3.6 shows the time consumed by the experiment. The time cost of each method includes test input generation time and mutation testing time. Dataset10000 and Dataset20000 have similar kill rates but with less time cost for Dataset10000. In the MLPBVA method, the test generation time includes training data preparation time, neural network training time, and MCMC computation time. The most time-consuming of these is the preparation time of the training data. For example, to prepare $20,000$ training data for the `triType` program, $20,000$ executions of the program are needed to extract execution path information to generate labels, with a time consumption of $11,000$ seconds. This makes the MLPBVA method more time-consuming than other methods. Therefore in order to reduce the time cost and improve the prediction accuracy of the neural network, we will consider using some structural coverage criteria in future research to help cover more parts of the code and thus obtain higher-quality training data instead of using random test input generation to generate training data.

Besides, there are many equal-conditional expressions in the programs such as `tcas`, and our currently proposed method is not good at generating exact data because the probability of generating exact data is low. We will also consider addressing this issue in future work.

Chapter 4

Optimal Test Input Generation for Boundary Value Analysis

4.1 Introduction

Evaluating the test coverage intending BVA is difficult since the boundaries are defined as like continuous values. Li and Miao [35] propose a series of model-based logic boundary coverage criteria, combining the boundary coverage criteria and the logic coverage criteria. Kosmatov et al. [36] focus on the development of boundary coverage criteria for test generation from formal models, formalize the boundary coverage criteria, the rational for the (BZ-TT) method and tool set. Utilizing the aforementioned techniques, it is necessary to have the specification that clearly states the boundary formally. However, many software development projects lack formal specifications in their development process. Identifying the exact boundaries within the input space of a software system can be complex.

Software applications often operate within multidimensional input domains. These domains might involve intricate data structures and interactions between various inputs. The complexity of these structures and interactions complicates the identification of boundaries, as they may not always align with conventional notions of limits or edges. Software behavior isn't always static. Dynamic behavior can lead to shifting or evolving boundaries, making it challenging to define and cover them adequately. The dynamic nature of software adds an extra layer of complexity in identifying and testing boundaries effectively. Complex data structures, interactions between inputs, and dynamic software behavior can complicate boundary identification.

Additionally, creating a comprehensive set of test inputs to cover all boundaries can be resource-intensive. As the number of input variables or dimensions increases, the number of potential boundary combinations can grow exponentially. This results in a combinatorial explosion of test inputs, requiring significant time and resources to create and execute. Achieving complete boundary coverage might not always be the most efficient or cost-effective strategy. Finding a balance between comprehensive testing and resource constraints is a challenge.

In this chapter, we attempt to define alternative measures, called boundary coverage distance (BCD). BCD introduces a metric that evaluates test inputs' quality concerning boundary coverage in BVA. It focuses on distance measurements, and considerations of lower and upper limits to gauge test inputs' proximity to the boundaries, ensuring comprehensive boundary coverage for more reliable software testing. In addition, based on BCD, we consider the optimal test input generation to minimize BCD under the random testing scheme. This method addresses the resource-intensive nature of generating exhaustive boundary tests by efficiently selecting critical test inputs that contribute the most to the BCD metric. In summary, the contributions of this chapter include: i) the definition of the boundary coverage distance to measure how much the test inputs cover the boundary; ii) developing test generation algorithms based on BCD.

4.2 Boundary Value Analysis

Boundary value analysis (BVA) is one of the most popular approaches to generating test inputs. It is empirically known that the probability of introducing bugs is higher around the program path changes, i.e. the boundary. BVA finds the boundary from specifications or programs and generates test inputs around the boundary.

Consider the formal definition of boundary. Let f and I be a software under test and an input domain of software, respectively. The output of software is defined as $O := \{y \mid y = f(x),\, x \in I\}$. Suppose that the output of software is divided into several categories. Let $O_1, \ldots, O_m$ be m-categorized outputs of software where $O = \cup_{i=1}^{m} O_i$ and $O_i \cap O_j = \phi$ for all $i \neq j$. The outputs in a category is considered to be equal in a sense. Then the input domain can be divided by the equivalent partitions, i.e.,

$$I_i := \{x \in I \mid y = f(x),\, y \in O_i\}. \tag{4.1}$$

Intuitively, the boundary is the input of software that crosses two equivalent partitions.

To define the boundary, we consider the functions that change the input of software. Let g be the bijection function from the input domain to the input domain; $g : I \to I$. Let G be a set of the functions with the following properties:

- For any $g \in G$, $g^{-1} \in G$ where g^{-1} is the inverse function of g.
- There exists at least one composite function of G from any $x \in I$ to any $y \in I$.

A function in G is regarded as a minimal operation that changes inputs. The first property corresponds to the existence of inverse operation. The second one means that any input in I can be generated by a chain of the operations. Therefore the boundary of equivalent partitions is given by

$$B_i := \{x \in I_i \mid \text{There exists } g \in G \text{ such that } y = f(g(x)),\, y \notin O_i\}. \tag{4.2}$$

Consider the software for judging whether the English examination passed. The English examination has two kinds of scores; listening and reading. Each of listening and reading is scored out of 100. The conditions to get the credit are (i) both listening and reading scores exceed 50, (ii) the total of listening and reading scores exceeds 120. The input of software is a pair of the listening and reading scores for a student, and the output is one of (1) 'the input is invalid', (2) 'the student gets the credit' and (3) 'the student does not get the credit'. For such software, we consider four functions to represent four operations changing inputs; (g_1) increasing the listening score by one, (g_2) decreasing the listening score by one, (g_3) increasing the reading score by one and (g_4) decreasing the reading score by one. In function set $G = \{g_1, g_2, g_3, g_4\}$, $(g_1)^{-1} = g_2$. The boundary sets are

$$\begin{aligned}
B_1 :=& \{(-1,0),\ldots,(-1,100),\\
& (101,0),\ldots,(101,100),\\
& (0,-1),\ldots,(100,-1),\\
& (0,101),\ldots,(100,101)\}, \qquad (4.3)\\
B_2 :=& \{(100,50),\ldots,(70,50),\\
& (50,100),\ldots,(50,70),\\
& (100,51),\ldots,(100,100),\\
& (51,100),\ldots,(99,100),\\
& (69,51),(68,52),\ldots,(52,68),(51,69)\}, \qquad (4.4)\\
B_3 :=& \{(0,0),(0,1),\ldots,(0,100),\\
& (100,0),\ldots,(100,49),\\
& (1,0),\ldots,(100,0),\\
& (0,100),\ldots,(49,100),\\
& (99,49),\ldots,(70,49),\\
& (49,99),\ldots,(49,70),\\
& (69,50),(68,51),\ldots,(51,68),(50,69)\}, \qquad (4.5)
\end{aligned}$$

where (x, y) means the scores for listening and reading, respectively. Fig. 4.1 shows the input domain and the boundaries of this software. The x-axis and y-axis correspond to the listening and reading scores, respectively. There are three equivalent partitions I_1, I_2 and I_3, and the boundaries are located on the edges of equivalent partitions. It should be noted that the input $(-1, -1)$ is not the boundary, because the input $(-1, -1)$ cannot be the input belonging to I_2 or I_3 even if we apply any operations.

Remark 1: Any operations can be used if it holds the two properties. For example, we can add the operations: (v) increasing both the listening and reading scores by one and (vi) decreasing both the listening and reading scores by one. In this case, the input $(-1, -1)$ becomes the boundary because $(0, 0)$ is generated by the operation (v) from the input $(-1, -1)$. In other words, the boundaries depend on the definitions of operations in the formal definition.

Remark 2: The outputs of software can be defined arbitrarily. If we focus on the execution paths on the program, the inputs $(49, 50)$ and $(50, 49)$ can be the different execution paths. In this example, we define the equivalent partitions only from the result of the judgment for the English examination by ignoring the execution paths, which is like a black-box approach. On the other hand, if the output of software, the behavior of software, is defined by the execution paths, as in a white-box approach, the equivalent partitions are changed. That is, the boundaries are also changed.

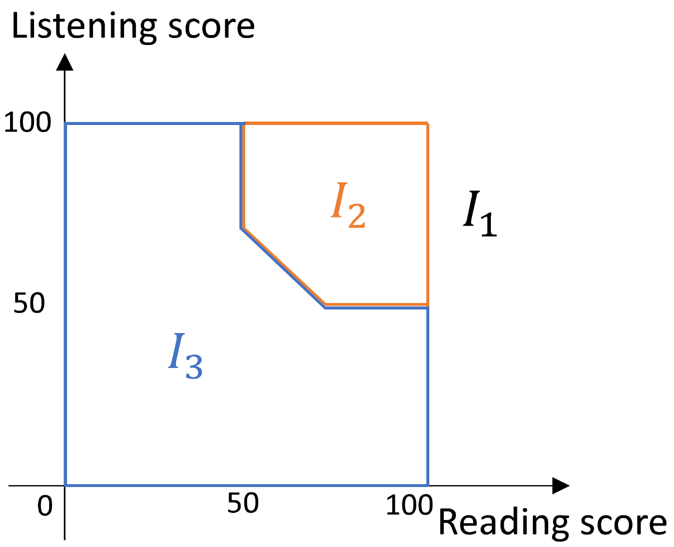


Figure 4.1: The input domain and the boundaries of the `English` software.

4.3 Boundary coverage distance

4.3.1 Definition

In this chapter, we propose a metric to evaluate the quality of test inputs from the perspective of BVA, called Boundary Coverage Distance (BCD). First we define the boundary coverage.

Definition (Boundary Coverage): Boundary coverage is the percentage of boundary values that are executed by a test suite.

Ideally, the boundary coverage is also achieved by a test suite to ensure highly reliable software. However, it is not easy to cover all the boundary values, because there are a huge number of boundary values and sometimes the number of boundary values becomes infinite. For example, even in the example of English examination, although it is a simple program, there are 963 boundary values.

Instead of the boundary coverage, we consider the distance from a given test suite to the test suite that achieves the boundary coverage. Let $d(x, y)$ be the function that returns the distance from x to y. Based on the formal definition of boundary, the distance is defined by the number of minimum operations from x to y, i.e,

$$d(x, y) = \min_n \{n \geq 0 \mid y = g_1(g_2(\cdots g_n(x))),\ g_1, \dots, g_n \in G\}. \tag{4.6}$$

For example, in English program mentioned in Section 2, the number of minimum operations from input $x = (-1, -1)$ to input $y = (0, 0)$ is 2.

Suppose that T_i is the test inputs (test suite) belonging to the equivalent partition I_i. The basic idea is the expansion of test inputs. The expansion means that each test input covers the test inputs that are within a given distance. The distance from T_i to I_i is defined as the minimum expansion distance for the test inputs in T_i until they cover all the boundary values B_i.

$$d(T_i, B_i) = \max_{y \in B_i} \min_{x \in T_i} d(x, y), \tag{4.7}$$

where $d(T_i, B_i) = \infty$ when T_i is empty.

Finally, the distance from the suite $S = \cup_{i=1}^m T_i$ to $B = \cup_{i=1}^m B_i$ is given by

$$d(T, B) = \max_{i=1,\dots,m} d(T_i, B_i). \tag{4.8}$$

We call this distance the boundary coverage distance (BCD). If BCD is small, the test inputs are placed close to the boundaries.

4.3.2 Computation of BCD

To compute BCD, we need to obtain all the boundary values. As mentioned before, it is not easy to get all the boundary values. Here we consider the lower and upper limits of BCD. Let $\underline{B}_i$ be a set of test inputs which are placed on B_i, i.e., $\underline{B}_i \subseteq B$. In this chapter, $\underline{B}_i$ is called the boundary points on I_i. From Eq. (4.21), it is clear that

$$d(T_i, \underline{B}_i) \leq d(T_i, B_i). \tag{4.9}$$

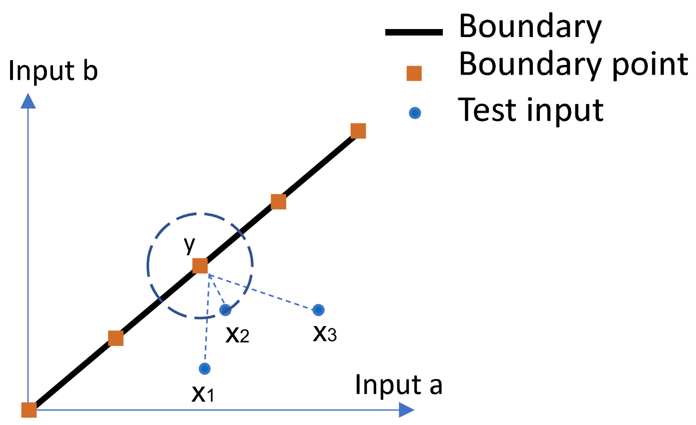


Figure 4.2: An example for cover a boundary point(Compared with x_1 and x_3, the test input x_2 can cover the boundary point y with the minimum distance $d(x_2, y)$).

Therefore we have the lower limit of BCD as follows.

$$\underline{\mathrm{BCD}}(T) = \max_{i=1,\ldots,m} d(T_i, \underline{B}_i). \tag{4.10}$$

Next we consider the upper limit of BCD. This chapter focuses on components of the boundary, called the boundary components. For example, if the boundary is represented by a line segment, the boundary components are line segments that are shorter than the original boundary. If the boundary is a plane, the boundary components are triangles that cover the plane. We assume that each boundary component is defined by a set of points. A line segment is represented by two start and end points. In the case of triangle, it is defined by a set of three points. In general, $\overline{B}_i := \{P_1, \ldots, P_k\}$ is a set of boundary components $P_1, \ldots, P_k$ that cover the boundary B_i where each boundary component consists of a set of points $P_i := \{x_1, \ldots, x_h\}$. The distance from a point x to a boundary component P_i is given by

$$d(x, P_i) = \max_{i=1,\ldots,h} d(x, x_i). \tag{4.11}$$

This gives us the distance from T_i to $\overline{B}_i$:

$$d(T_i, \overline{B}_i) = \max_{p\in\overline{B}_i} \min_{x\in T_i} d(x, p). \tag{4.12}$$

Since Eq. (4.11) takes the maximum of points on the boundary, the distance of

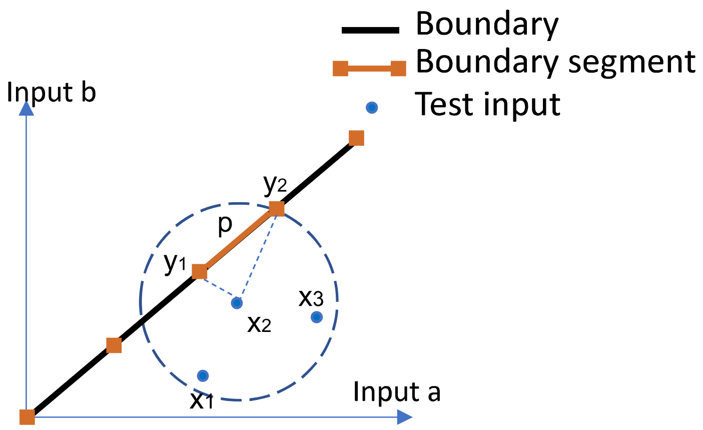


Figure 4.3: An example for cover a boundary component (segment)(The test input x_2 can cover the boundary segment p with a distance of $d(x_2, y_2)$.

the boundary components is greater than the exact distance, i.e.,

$$d(T_i, B_i) \leq d(T_i, \overline{B}_i). \tag{4.13}$$

The upper limit of BCD becomes

$$\overline{\text{BCD}}(T) = \max_{i=1,\ldots,m} d(T_i, \overline{B}_i). \tag{4.14}$$

Since both $\underline{B}_i$ and $\overline{B}_i$ consist of the points that are the subset of B_i, they can be computed without extracting all the boundary values.

For instance, we select the following subsets of B_1, B_2 and B_3 in the example:

$$\begin{aligned}\underline{B}_1 :=&\{(-1,0),(-1,100),(101,0),(101,100),\\ &(0,-1),(100,-1),(0,101),(100,101)\}, &(4.15)\\ \underline{B}_2 :=&\{(100,50),(70,50),(50,100),(50,70),(100,100)\}, &(4.16)\\ \underline{B}_3 :=&\{(0,0),(0,100),(100,0),(49,100),(100,49),(49,70),(70,49)\}. &(4.17)\end{aligned}$$

$$\begin{aligned}\overline{B}_1 :=&\{\{(-1,0),(-1,100)\},\{(101,0),(101,100)\},\\ &\{(0,-1),(100,-1)\},\{(0,101),(100,101)\}\}, &(4.18)\\ \overline{B}_2 :=&\{\{(100,50),(70,50)\},\{(50,100),(50,70)\},\{(100,50),(100,100)\},\\ &\{(50,100),(100,100)\},\{(70,50),(50,70)\}\}, &(4.19)\\ \overline{B}_3 :=&\{\{(0,0),(0,100)\},\{(0,0),(100,0)\},\end{aligned}$$

$$\{(0,100),(49,100)\},\{(100,0),(100,49)\},$$
$$\{(49,100),(49,70)\},\{(100,49),(70,49)\},$$
$$\{(49,70),(70,49)\}\}. \tag{4.20}$$

It should be noted that $\underline{B}_i = \cup_{c\in\overline{B}_i} c$. Fig. 4.2 shows the boundary points and Fig. 4.3 the boundary components (segments).

4.4 Test Input Generation

In this section, we consider the test input generation that minimizes BCD. Suppose that the boundary points $\underline{B}_i$ and the boundary components $\overline{B}_i$ are given. In this situation, we obtain additional n test inputs minimizing BCD. Ideally, when the BCD is reduced to a minimum value, the n test inputs to be optimized move to the boundary point or to the center of the boundary component. Since a large number of test inputs cause a large amount of cost on software testing. In this study, we first empirically analyze all possible boundary values as boundary points. Then the boundary region to be tested is selected by minimizing the value of the BCD of the n test inputs.

We propose three test input generation algorithms that are very similar to MCMC (Markov Chain Monte Carlo) method [26].

The optimization process is shown in Algorithm 1, Algorithm 2 and Algorithm 3, respectively.

Algorithm 1 first randomly generates n test inputs from the input domain

Algorithm 1 An algorithm to generate boundary test inputs by reducing BCD

```
1: $Initial\_test\_set$ ←{randomly generate n test inputs from the input domain}
2: while $j \le Iter$ do
3:     $BCD \leftarrow BCD(Initial\_test\_set)$
4:     $t \leftarrow randomly\ select\ a\ test\ input\ from\ Initial\_test\_set$
5:     $t' \leftarrow Q(t';t)$
6:     $Candidate\_test\_set$ ← {Replace the t in the Initial_test_set with the can-
       didate t'}
7:     $BCD' \leftarrow BCD(candidate\_test\_set)$
8:     if $BCD' < BCD$ then
9:         $Initial\_test\_set \leftarrow candidate\_test\_set$
10:    end if
11:    $j \leftarrow j + 1$
12: end while
```

Algorithm 2 An algorithm that considers each boundary point or each boundary component

```
1: $Initial\_test\_set$ ←{randomly generate n test inputs from the input domain}
2: while $j \leq Iter$ do
3:     $Calculate\ d(T_i, y)$
4:     $t \leftarrow randomly\ select\ a\ test\ input\ from\ Initial\_test\_set$
5:     $t' \leftarrow Q(t'; t)$
6:     $Candidate\_test\_set$ ← {Replace the t in the Initial_test_set with the candidate t'}
7:     $Calculate\ d'(T_i, y)$
8:     if $b\_decrease > 0$ and $b\_increase = 0$ then
9:         $Initial\_test\_set \leftarrow candidate\_test\_set$
10:    end if
11:    $j \leftarrow j + 1$
12: end while
```

Algorithm 3 Accept with probability

```
1: $Initial\_test\_set$ ← {randomly generate n test inputs from the input domain}
2: while $j \leq Iter$ do
3:     $Calculate\ d(T_i, y)$
4:     $t \leftarrow randomly\ select\ a\ test\ input\ from\ Initial\_test\_set$
5:     $t' \leftarrow Q(t'; t)$
6:     $Candidate\_test\_set$ ← {Replace the t in the Initial_test_set with the candidate t'}
7:     $Calculate\ d'(T_i, y)$
8:     $Generate\ a\ uniform\ random\ number\ U$
9:     if $\frac{b\_decrease}{b\_decrease + b\_increase} > U$ then
10:        $Initial\_test\_set \leftarrow candidate\_test\_set$
11:    end if
12:    $j \leftarrow j + 1$
13: end while
```

as an initial test set. Then the initial test set is optimized by reducing the BCD value (lines 2-10). During the optimization process, the algorithm 1 first calculates the BCD of the initial test set. Then it randomly selects a test input t from the initial test set, and generates a new candidate t' according to the proposal distribution, provided that t is given $t' \sim Q(t';t)$. If replacing t with the candidate t' can reduce the value of BCD, the candidate t' is accepted and replaces t in the initial test set, otherwise, the candidate t' is rejected. After the optimization process performs a fixed number of iterations, the initial test set is moved to the boundary. In the algorithm 1, BCD can be computed as $\underline{BCD}$ or $\overline{BCD}$ mentioned in section 4.3.2, or even as the mean of $\underline{BCD}$ and $\overline{BCD}$, denoted as $BCD\_mean = mean(\underline{BCD} + \overline{BCD})$.

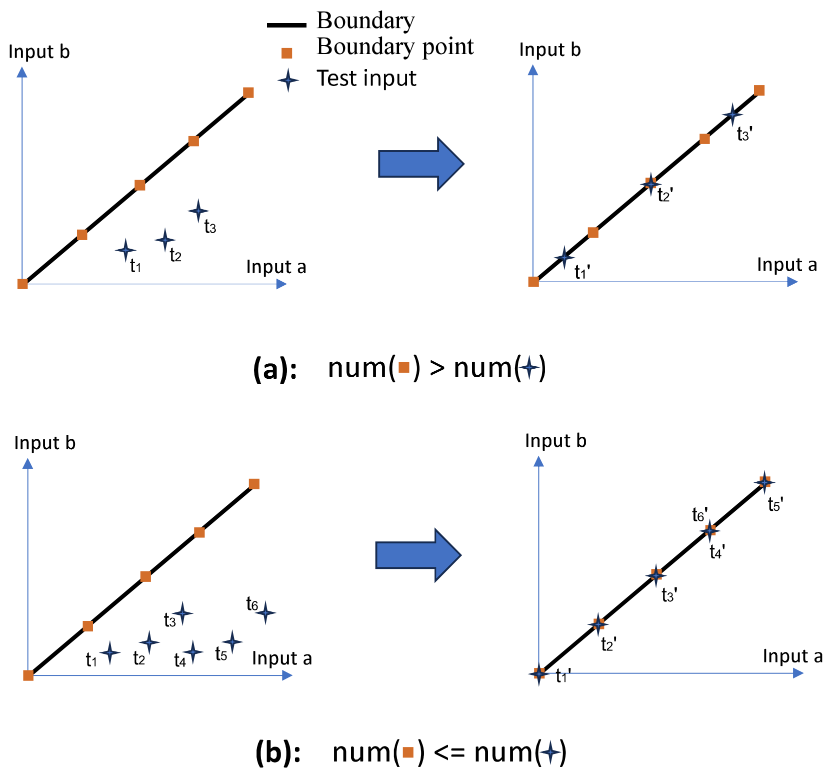


Figure 4.4: The ideal solutions generated by our algorithms when applying the $\underline{BCD}(\underline{B_i})$ criteria

Algorithm 1 only accepts candidates that can reduce the maximum distance among the minimum distances between the boundary point (or boundary line

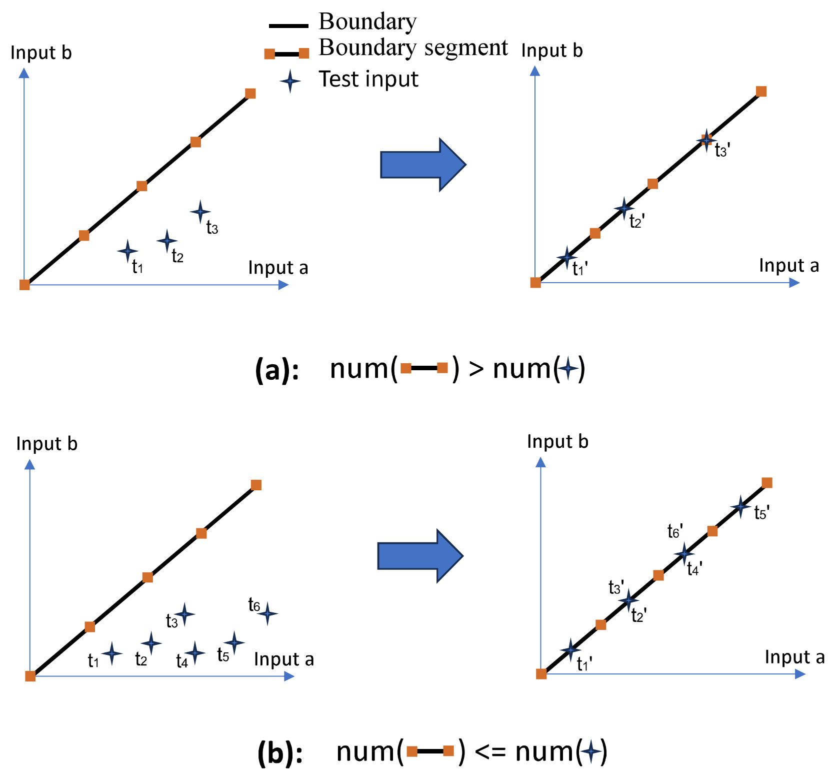


Figure 4.5: The ideal solutions generated by our algorithms when applying the $\overline{BCD}(\overline{B_i})$ criteria

segment) and the test input. This means that the optimization goal of each iteration is only a boundary point or a boundary component, and the optimization process is slow.

With this in mind, we judged whether to accept the candidate by directly comparing the coverage distance $d(T_i, y)$ for each boundary point or boundary component.

$$d(T_i, y) = \min_{x \in T_i} d(x_i, y) \quad for \quad y \in B_i. \tag{4.21}$$

Let *b_decrease* be the number of boundary points or boundary components whose coverage distance is decreased by the candidate test set. Similarly, *b_increase* denotes the number of boundary points or boundary components whose coverage distance is increased by the candidate test set. We then get two algorithms for generating boundary test inputs, shown in Algorithm 2 and

Table 4.1: Combining the three algorithms with the three methods of BCD calculation results in seven approaches.

Methods ╲ BCD / Algorithms	$\underline{BCD}(\underline{B_i})$	$\overline{BCD}(\overline{B_i})$	BCD_mean
Algorithm1	A1-$\underline{BCD}$	A1-$\overline{BCD}$	A1-BCD_mean
Algorithm2	A2-$\underline{B_i}$	A2-$\overline{B_i}$	╲
Algorithm3	A3-$\underline{B_i}$	A3-$\overline{B_i}$	╲

Algorithm 3 respectively. Algorithm 2 accepts a candidate when the candidate can reduce the coverage distance of one or more boundary points (or one or more boundary components) without increasing the coverage distance of any others. However, in Algorithm 3, even if the candidate increases the coverage distance of some boundary points (or some boundary components), it may be accepted with a certain probability. In Algorithms 2 and 3, *b_decrease* and *b_increase* can be calculated with $\overline{B_i}$ or $\underline{B_i}$. Combining the three algorithms and the three BCD calculation methods, 7 methods can be obtained, as shown in Table 4.1.

Fig. 4.4 and Fig. 4.5 demonstrate the ideal solutions generated by our algorithms when applying the $\underline{BCD}(\underline{B_i})$ and $\overline{BCD}(\overline{B_i})$ criteria, respectively. For the purpose of illustration, we assume a scenario where five boundary points are evenly spaced along a linear boundary, with a unit distance between each pair of adjacent points. The ideal solution varies based on the number of boundary points and test inputs. Fig. 4 (a) represents a scenario where the number of test inputs is fewer than the boundary points. Here, one test input is positioned at the central boundary point, while the remaining test inputs are placed midway between adjacent boundary points. This arrangement results in an optimal $\underline{BCD}$ value of 0.5. Conversely, as shown in Fig. 4 (b), when the number of test inputs exceeds the boundary points, each test input aligns with a boundary point, achieving an optimal $\underline{BCD}$ value of 0. Fig. 4.5 illustrates the ideal solutions under the $\overline{BCD}$ criterion, which shows a different pattern compared to the $\underline{BCD}$ criterion. If the number of boundary points is greater than the test inputs, some test inputs are likely to align with the boundary points themselves. However, when there are more test inputs than boundary points, all test inputs tend to be the middle of the boundary segments. The average outcomes of the $\underline{BCD}$ and $\overline{BCD}$ methods demonstrate these intermediate tendencies. These scenarios represent ideal states in a highly simplified boundary context. It is important to note that in practical applications with more complex boundaries, the arrangement of test inputs is likely to be more intricate.

4.5 Experiment

This section presents experiments to investigate the fault detection capabilities of the three algorithms of the BCD approach and compare them with RT, ART and concolic testing. RT, ART and concolic testing are described in Chapter 3.4.1 We conducted experiments to generate test inputs for the previously mentioned English examination program and four real programs.

4.5.1 Programs under Testing

The method we have proposed is designed for generating boundary test inputs in software testing by optimizing a set of test inputs to move towards boundary points. This approach is suitable for programs where inputs can transition from one state to another through measurable operations. The distance between two test inputs, in terms of the number of operational steps required for transformation, determines the applicability of this method.

We select four programs used in the existing literature and English examination program to evaluate the effect of our approach. The descriptions of these subject programs are shown in Table 4.2. And the details about all 5 programs are shown in Tables 4.3 and 4.4, such as the dimensional number of program inputs, the range of input domain, the number of patterns, line of code (LOC), mutation faults information, and the number of boundary points (num_Bpoint).

Table 4.2: Experimental programs

Prog Name	Description
triType [29]	The type of a triangle
nextDate [30]	Calculate the following date of the given day
findMiddle [31]	Find the middle number among three numbers
English	Judge whether the English examination passed
miniSAT	Minimalistic SAT solver

Table 4.3: Details of subject programs

Program	Dim	Input Domain		Test patterns	Size(LOC)
		From	To		
findMiddle	3	(-10, -10, -10)	(10, 10, 10)	9.26×10^3	36
English	2	(0,0)	(100, 100)	1.02×10^4	26
triType	3	(0, 0, 0)	(50, 50, 50)	1.32×10^5	41
nextDate	3	(0, 0, 0)	(2018, 12, 31)	8.39×10^5	90
miniSAT	9	(1,1,1,1,1,1,1,1,1) (-1,-1,-1,-1,-1,-1,-1,-1,-1)	(5,5,5,5,5,5,5,5,5) (-5,-5,-5,-5,-5,-5,-5,-5,-5)	1.0×10^9	1069

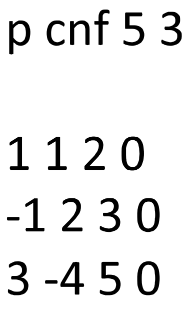
Table 4.4: Mutation faults information and the number of boundary points.

Program	Fault types					Total Faults	num_Bpoint
	OBOB	ROR	AOR	SVR	LOR		
triType	6	6		3	6	21	420
nextDate	16	7			8	31	685
findMiddle	15	14			5	34	1092
English	14	7	2		2	25	283
miniSAT	8	16	2		9	35	279

For programs other than miniSAT, we define the minimum operation as plus one and its opposite operation as minus one. In miniSAT [37] experiment, we tested the solver.cc module in miniSAT version 2.2. The problem(Test input) is fixed as a 3-SAT problem with 5 variables and 3 clauses. This is represented in the DIMACS CNF format as Fig. 4.6. There are a total of 10^9 possible test patterns for this problem. In this problem we define two operations Change(x, y) and Pos(x, y). The operation "Change(x, y)" allows for increasing the numbers x and y (ranging from 1 to 5) in the clauses. The operation "Pos(x, y)" is used to make the y-th number in the x-th clause a positive value. For example, converting -1 to 1. There are a total of 18 types of operations, consisting of 9 "Change" operations and 9 "Pos" operations. Each operation corresponds to how many times it is applied. For example, the vector representing the number of operations needed to change from test input A to test input B might look like this:

$$[0, 1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0] \tag{4.22}$$

In this vector, the "-1" signifies the inverse operation, which means either "decrease the variable number by one" or "make it negative." The absolute value of -1 corresponds to the number of times the operation is applied. In this context,



```
p cnf 5 3

1 1 2 0
-1 2 3 0
3 -4 5 0
```

Figure 4.6: DIMACS CNF format.

the "distance" between test inputs A and B is determined by the total number of operations needed to transform one into the other. For example, if there are two "-1" operations, the distance between test input A and test input B is 2.

In this chapter, boundary points are obtained by manual analysis of the source code. First, the input domain is divided into m equivalent partitions based on the output. Then, in each equivalent partition, boundary points are generated based on the definition in Section 4.2.

4.5.2 Design Parameters

In the experiment, we investigate the fault detection capabilities of RT, ART, BCD-Algorithm1(A1), BCD-Algorithm2(A2), and BCD-Algorithm3(A3). First, the input domains of programs triType, nextDate, findMiddle, English, and miniSAT are divided into 5, 15, 3, 3, and 2 equivalent partitions, respectively. Then, the RT randomly generates test inputs for each equivalent partition. Specifically, 10 test inputs are generated for each partition of triType, 3 for each partition of nextDate, 10 for each partition of findMiddle, 10 for each partition of English, and 15 for each partition of miniSAT. In total, 50 test inputs are generated for the program triType, 45 for nextDate, 30 for findMiddle, 30 for English, and 30 for miniSAT.

We then use A1, A2, and A3 to optimize the RT-generated test set. To generate the candidate, we use the uniform distribution as the proposal distribution. And the number of iterations for the optimization process is set to 10000.

Table 4.5: Kill rates for the test sets generated by different methods

Method	Kill rate				
	triType (n=50)	nextDate (n=45)	findMiddle (n=30)	English (n=30)	miniSAT (n=30)
RT	0.66	0.54	0.61	0.36	0.65
ART	0.61	0.51	0.63	0.44	0.64
A1_$\underline{BCD}_{max}$	0.8	0.7	0.61	0.44	0.65
A1_$\underline{BCD}_{mean}$	0.95	1	0.97	0.8	0.94
A1_$\overline{BCD}_{max}$	0.8	0.7	0.73	0.44	0.65
A1_$\overline{BCD}_{mean}$	0.9	0.93	0.88	0.68	0.68
A1_BCD_mean_{max}	0.61	0.74	0.94	0.52	0.59
A1_BCD_mean_{mean}	0.71	0.96	0.94	0.71	0.65
A2_point	0.85	0.96	0.76	0.52	0.85
A2_seg	0.8	0.9	0.85	0.59	0.62
A3_point	0.76	0.93	0.55	0.44	0.62
A3_seg	0.76	0.83	0.94	0.4	0.59
KLEE	0.7 (n=14)	0.9 (n=56)	0.88(n=13)	0.6(n=8)	0(n=1)

4.5.3 Result

Table 4.5 shows the kill rates for the test sets generated by different methods. KLEE generates n test inputs for each of the four programs, as shown in the Table 4.5. In Algorithm1, BCD can be computed as $\underline{BCD}$, $\overline{BCD}$ and BCD_mean. In the definition of the BCD calculation, we use the max operation to calculate the boundary coverage distance. In this experiment, we also used an alternative method to replace all max operations in the BCD calculation process with $mean$ operations, so that each boundary point affects the BCD calculation results. Under the max operation or $mean$ operation, we denote the BCD calculation method of Algorithm1(A1) as A1_$\underline{BCD}_{max}$, A1_$\underline{BCD}_{mean}$, A1_$\overline{BCD}_{max}$, A1_$\overline{BCD}_{mean}$,A1_BCD_mean_{max} and A1_BCD_mean_{mean}, respectively. In the Table 4.5, the A2_point method uses Algorithm 2 to optimize the test set based on each boundary point, and the A2_seg method optimizes the test set based on each boundary segment.

From the results, it can be seen that the most BCD-based methods have better fault detection ability than RT and ART, and the kill rate of test inputs generated by BCD-based methods is better than that of concolic testing, because concolic testing is not good at detecting OBOB bugs. In the experiment of using the KLEE tool to generate test inputs for the miniSAT program, we used KLEE

Table 4.6: Accept rate during optimization of each method.

Method	Accept rate				
	triType	nextDate	findMiddle	English	miniSAT
A1_$\underline{BCD}_{max}$	0.0026	0.0006	0	0.0003	0.0001
A1_$\underline{BCD}_{mean}$	0.0123	0.0115	0.0075	0.0081	0.00075
A1_$\overline{BCD}_{max}$	0.0025	0.0009	0.0009	0.0021	0.0003
A1_$\overline{BCD}_{mean}$	0.0151	0.0101	0.0075	0.0083	0.0073
A1_BCD_mean_{max}	0.0035	0.0011	0.0024	0.004	0.0006
A1_BCD_mean_{mean}	0.0133	0.0098	0.007	0.0076	0
A2_point	0.0043	0.0039	0.0016	0.0022	0.0023
A2_seg	0.0053	0.0031	0.0012	0.0017	0.0035
A3_point	0.1807	0.2369	0.4172	0.3815	0.135
A3_seg	0.1655	0.195	0.4125	0.3875	0.0748

to provide a symbolic file, and KLEE only generated one test input. It can be seen that it is difficult to test miniSAT using symbolic methods. Meanwhile, test inputs generated by A1_$\underline{BCD}_{mean}$ can kill more mutations than other methods. For the same number of iterations, A1 with BCD_{mean} accepts more candidates than A1 with BCD_{max}, as shown in Table 4.6. Therefore, reducing the BCD computed with the mean operation is more helpful for test set optimization than the max operation. However, the acceptance rate of A2 and A3 is slightly higher than that of A1_BCD_{mean}, but due to the large number of boundary points, judging the coverage distance of each boundary point does not give a good result.

Fig. 4.7 shows the distribution of the input data generated by RT and the optimization results of A1_$\underline{BCD}_{mean}$ on the test set generated by RT in the English experiment. From the distribution graph, it can be seen intuitively that most of the input data in the optimized test set moved to the boundary.

Tables 4.7 and 4.8 show the time cost of the experiment, including the time to generate the test input (optimization time) and the mutation time. Since in the ART method, the detection of injected faults (mutation testing) is performed during the test input generation process, the time cost of the ART method is not presented in table. the ART time costs of programs triType, nextDate, findMiddle, English, and miniSAT are 248, 572, 193, 253, and 996 seconds, respectively. Although the time cost of the BCD-based method is higher than

Table 4.7: Time cost of test input generation.

Method	Test input generation time (sec)				
	triType	nextDate	findMiddle	English	miniSAT
RT	0.002	0.002	0.002	0.001	0.002
A1_$\underline{BCD}_{max}$	253	769	597	236	134
A1_$\underline{BCD}_{mean}$	242	799	295	154	142
A1_$\overline{BCD}_{max}$	432	1163	921	319	359
A1_$\overline{BCD}_{mean}$	421	1194	1043	316	382
A1_BCD_mean_{max}	442	1559	966	338	410
A1_BCD_mean_{mean}	442	1500	1004	336	404
A2_point	244	544	303	144	151
A2_seg	390	541	864	298	365
A3_point	241	510	270	158	146
A3_seg	389	534	830	309	387
KLEE	1	1	1	1	1

other methods, the BCD-based method can generate better quality test inputs and detect more faults.

Table 4.8: Time cost of mutation testing.

Method	Mutation time (sec)				
	triType	nextDate	findMiddle	English	miniSAT
RT	291	388	256	196	1232
A1_$\underline{BCD}_{max}$	282	440	242	195	1133
A1_$\underline{BCD}_{mean}$	287	420	265	201	1324
A1_$\overline{BCD}_{max}$	295	392	269	196	1102
A1_$\overline{BCD}_{mean}$	284	388	297	189	1309
A1_BCD_mean_{max}	285	420	273	198	1190
A1_BCD_mean_{mean}	296	405	291	198	1114
A2_point	287	457	259	198	1266
A2_seg	302	399	250	197	1308
A3_point	284	346	246	198	1146
A3_seg	284	359	243	212	1125
KLEE	68	469	67	56	43

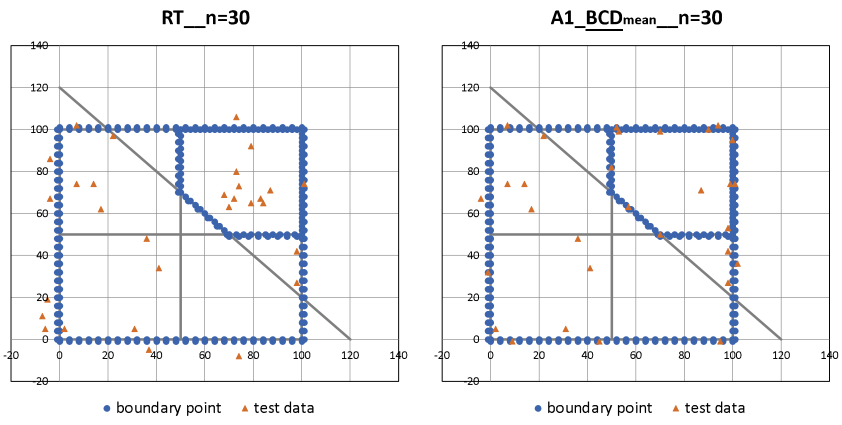


Figure 4.7: The input data distribution generated by RT and A1_$\underline{BCD}_{mean}$ in the English experiment.

Chapter 5

Conclusion

This dissertation contains four folds, around the generation of test inputs in software testing, we discussed the automation of test coverage achievement and boundary value analysis, respectively. From the perspective of branch coverage, we propose a strategy to enhance branch coverage using a GAN model. In addition, to improve the fault detection ability of test inputs, we consider the use of machine learning techniques to automate Boundary Value Analysis (BVA) for generating boundary test inputs. We also introduce a new metric for boundary coverage.

In Chapter 2, we introduced an automatic test input generation framework based on Generative Adversarial Network (GAN), aim at achieving full branch coverage. In the framework, we used not only the test inputs but also the corresponding execution path as inputs of GAN. The generator of GAN was used to generate test input and the corresponding expected path. According to the path information, we selected the test input that may cover the new branch. The discriminator was used to reinforce the generation ability of the generator so that the generator could generate inputs that conforms to the true distribution. In this way, our coverage improvement strategy was meaningful. We conducted the experiments to examine the effectiveness of our proposed framework. The results show that GAN-based methods can automatically generate test inputs for programs with large-scale branches to increase coverage. Compared with the standard GAN and BiGAN, WGAN-GP provided the better performance in the experiment, and compared with the integration testing application, our framework was more effective in the unit test scheme. Besides, the convolution neural network did not show better prediction performance than the fully connected neural network. Using GAN to generate test inputs has a technical problem that some branches are difficult to cover. It can be divided into the following three categories. The first is "==" conditional branch. It is difficult for GAN to generate a specific value, especially the type of program input is float. The second is the branch with complex conditional expression. When the gap between the target data and the training data is large, it is also difficult to cover it with the GAN method. In order to achieve full branch coverage, we need to do more work to tackle the above problem. In addition, since the execution path depends on the control flow of the program, in future research we will consider converting the control flow graph into the form of a graph neural network and incorporating it into the GAN model. Providing structured path information of the program to the GAN, which would improve the ability to learn path distributions. In practical applications, the GAN method can be applied in any test scenario that focuses on branch coverage, as long as the path information containing branch execution information can be extracted.

In Chapter 3, the MLP based approach to automatically generate test inputs with Boundary Value Analysis was discussed. First, we train an MLP-based discriminator that determines whether two test inputs have the same execution path or not. Second, we create test inputs based on Markov Chain Monte Carlo (MCMC) using the discriminator's outputs. We conducted a set of experiments on a simple program and seven real programs to exhibit the performance of MLP-based approach. Our results showed that the MLP-based approach could generate test inputs close to the boundary for testing and has better fault detection ability than RT and ART. Besides, the MLP-based method outperforms the manually-performed boundary analysis in four of the seven real programs tested, and outperforms the concolic testing in three of the seven real programs tested. The accuracy of neural network predictions largely affects the performance of the MLP-based approach. In order to use neural networks to generate higher-quality test inputs, we need to consider improving the performance of neural networks in the future, such as employing the Large Language Model (LLM) to predict program execution paths by learning relationships between code blocks and understanding conditional statements and loops, so that neural networks can better learn the boundary information of programs. At the same time, high-quality training data will also improve the prediction accuracy of the neural network, and can greatly reduce the time cost, so we will also improve the training data generation strategy in future work. Besides, there are many equal-conditional expressions in the programs, and our currently proposed method is not good at generating exact data. To improve the fault detection ability of our proposed method, this problem needs to be solved in our future work. We will also consider applying our method to complex software systems with large input spaces and complex and highly structured inputs. In the experiment, we only tested the C language. In practical applications, for other languages, we can apply this method by simply changing the way path extraction is performed.

In Chapter 4, we proposed a new boundary coverage metric called Boundary Coverage Distance (BCD). Then a set of randomly generated test inputs is optimized based on BCD to generate boundary values. We conducted the experiments on four programs to exhibit the performance of the BCD-based method. Our results showed that the BCD-based method can generate test inputs close to the boundary and can increase the fault detection rate of a randomly generated test set. The selection of the boundary points significantly affects the performance of the BCD-based method. In this paper, we analyze the boundaries of several programs to obtain boundary points. However, in the case of large programs or complex numerical computation programs, the identification of the boundary becomes difficult and poses a significant problem. In this study, we manually analyze and obtain the boundary points. To improve the automation of our method, we will explore possibilities such as integrating existing analysis tools or using deep learning techniques to predict boundary points. To generate test input for software testing, it is important to consider different types of input. If the program input is numerical data, such as continuous data, we can use Gaussian distribution or similar techniques to infer neighboring points. Conversely, for discrete data, simple addition and subtraction operations can be used to determine adjacent points. However, when the program input consists of non-numerical data, such as an array (e.g., in a sorting problem) or a string of letters, the definition of boundaries and adjacent inputs requires further consideration for future optimization. In practical applications, the BCD can be applied in any test scenario that focuses on branch coverage, as long as the path information containing branch execution information can be extracted. In practical applications, when we know the boundary information, we can use BCD to measure whether a set of test inputs covers all boundaries. Conversely, we can also use the BCD-based generation algorithm we proposed to select which parts of the boundary we need to focus on testing.

The main contributions of this dissertation are proposing several methods for generating high-quality test inputs. Initially, we explore the application of the GAN model to automatically generate test inputs, aiming to achieve full branch coverage. Subsequently, we introduce a MLP-based approach for the automated implementation of Boundary Value Analysis (BVA). Expanding on these topics, we delve into the discussion of a novel boundary coverage metric. Through extensive experiments, the efficacy of these methods is demonstrated, including improvements in branch coverage and fault detection capabilities.

Bibliography

[1] Meysam Valueian, Niousha Attar, Hassan Haghighi, and M Vahidi-Asl. Constructing automated test oracle for low observable software. *Scientia Iranica*, 27(3):1333–1351, 2020.

[2] Abhishek Singhal, Abhay Bansal, et al. Generation of test oracles using neural network and decision tree model. In *2014 5th International Conference-Confluence The Next Generation Information Technology Summit (Confluence)*, pages 313–318. IEEE, 2014.

[3] Seyed Reza Shahamiri, Wan Mohd Nasir Wan Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. An automated framework for software test oracle. *Information and Software Technology*, 53(7):774–788, 2011.

[4] Seyed Reza Shahamiri, Wan MN Wan-Kadir, Suhaimi Ibrahim, and Siti Zaiton Mohd Hashim. Artificial neural networks as multi-networks automated test oracle. *Automated Software Engineering*, 19:303–334, 2012.

[5] Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *Acm computing surveys (csur)*, 29(4):366–427, 1997.

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

[7] Gordon Fraser and Andrea Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical software engineering*, 20:611–639, 2015.

[8] Yuan Zhan and John A Clark. A search-based framework for automatic testing of matlab/simulink models. *Journal of Systems and Software*, 81(2):262–285, 2008.

[9] Mark Harman, Yue Jia, and Yuanyuan Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12. IEEE, 2015.

[10] Felix Dobslaw, Francisco Gomes de Oliveira Neto, and Robert Feldt. Boundary value exploration for software analysis. In *2020 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 346–353. IEEE, 2020.

[11] Stuart C Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings Fourth International Software Metrics Symposium*, pages 64–73. IEEE, 1997.

[12] Bingchiang Jeng and István Forgács. An automatic approach of domain test data generation. *Journal of Systems and Software*, 49(1):97–112, 1999.

[13] Ruilian Zhao, Michael R Lyu, and Yinghua Min. Automatic string test data generation for detecting domain errors. *Software Testing, Verification and Reliability*, 20(3):209–236, 2010.

[14] Shaukat Ali, Tao Yue, Xiang Qiu, and Hong Lu. Generating boundary values from ocl constraints using constraints rewriting and search algorithms. In *2016 IEEE Congress on Evolutionary Computation (CEC)*, pages 379–386. IEEE, 2016.

[15] Robert Feldt and Felix Dobslaw. Towards automated boundary value testing with program derivatives and search. In *Search-Based Software Engineering: 11th International Symposium, SSBSE 2019, Tallinn, Estonia, August 31–September 1, 2019, Proceedings 11*, pages 155–163. Springer, 2019.

[16] Zhiqiang Zhang, Tianyong Wu, and Jian Zhang. Boundary value analysis in automatic white-box test generation. In *2015 IEEE 26th International*

Symposium on Software Reliability Engineering (ISSRE), pages 239–249. IEEE, 2015.

[17] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.

[18] Yongjun Hong, Uiwon Hwang, Jaeyoon Yoo, and Sungroh Yoon. How generative adversarial networks and their variants work: An overview. *ACM Computing Surveys (CSUR)*, 52(1):1–43, 2019.

[19] Gcov: https://gcc.gnu.org/onlinedocs/gcc/gcov.html.

[20] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. Improved training of wasserstein gans. *Advances in neural information processing systems*, 30, 2017.

[21] Jeff Donahue, Philipp Krähenbühl, and Trevor Darrell. Adversarial feature learning. *arXiv preprint arXiv:1605.09782*, 2016.

[22] Martin Arjovsky, Soumith Chintala, and Léon Bottou. Wasserstein generative adversarial networks. In *International conference on machine learning*, pages 214–223. PMLR, 2017.

[23] Gsl: https://www.gnu.org/software/gsl/doc/html/intro.html.

[24] Bo Zhou, Hiroyuki Okamura, and Tadashi Dohi. Enhancing performance of random testing through markov chain monte carlo methods. *IEEE Transactions on Computers*, 62(1):186–192, 2011.

[25] Stephen Brooks. Markov chain monte carlo method and its application. *Journal of the royal statistical society: series D (the Statistician)*, 47(1):69–100, 1998.

[26] Siddhartha Chib and Edward Greenberg. Understanding the metropolis-hastings algorithm. *The american statistician*, 49(4):327–335, 1995.

[27] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[28] Tsong Yueh Chen, Hing Leung, and Ieng Kei Mak. Adaptive random testing. In *Advances in Computer Science-ASIAN 2004. Higher-Level Decision Making: 9th Asian Computing Science Conference. Dedicated to Jean-Louis Lassez on the Occasion of His 5th Birthday. Chiang Mai, Thailand, December 8-10, 2004. Proceedings 9*, pages 320–329. Springer, 2005.

[29] Nicky Williams, Bruno Marre, Patricia Mouy, and Muriel Roger. Pathcrawler: Automatic generation of path tests by combining static and dynamic analysis. In *European Dependable Computing Conference*, pages 281–292. Springer, 2005.

[30] Zeina Awedikian, Kamel Ayari, and Giuliano Antoniol. Mc/dc automatic test input data generation. In *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*, pages 1657–1664, 2009.

[31] Ghani K. Searching for test data. *Ph. D Thesis*, 2009.

[32] Saul A Teukolsky, Brian P Flannery, WH Press, and WT Vetterling. Numerical recipes in c. *SMR*, 693(1):59–70, 1992.

[33] Hyunsook Do, Sebastian Elbaum, and Gregg Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10:405–435, 2005.

[34] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[35] Liping Li and Huaikou Miao. Model-based boundary coverage criteria for logic expressions. *Appl. Math*, 6(1S):31S–34S, 2012.

[36] Nikolai Kosmatov, Bruno Legeard, Fabien Peureux, and Mark Utting. Boundary coverage criteria for test generation from formal models. In *15th International Symposium on Software Reliability Engineering*, pages 139–150. IEEE, 2004.

[37] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *International conference on theory and applications of satisfiability testing*, pages 502–518. Springer, 2003.

Publication List of the Author

Publications in this dissertation

[J-1] X. Guo, H. Okamura and T. Dohi, "Automated Software Test Data Generation With Generative Adversarial Networks," IEEE Access 10 (2022): 20690-20700

[J-2] X. Guo, H. Okamura and T. Dohi, "Optimal Testcase Generation for Boundary Value Analysis," Software Quality Journal(accepted).

Referred Conferences

[C-1] X. Guo, H. Okamura and T. Dohi, "Towards Automated Software Testing with Generative Adversarial Networks," in Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks - Supplemental Volume (DSN-S), Taipei, Taiwan, 2021, pp. 21-22, doi: 10.1109/DSN-S52858.2021.00021.

[C-2] X. Guo, H. Okamura and T. Dohi, "Improvement of MCMC Random Testing with Neural Networks," in Proceedings of the 5th International Conference on Mathematical Techniques in Engineering Applications(ICMTEA 2021), 2021, 10 pages.

[C-3] X. Guo, H. Okamura and T. Dohi, "A Note on Finding Boundary Values of Programs with Neural Networks," in Proceedings of the 10th Asia Pacific International Symposium on Advanced Reliability and Maintenance Modeling (APARM 2022), 2022, 5 pages.

[C-4] X. Guo, H. Okamura and T. Dohi, "Towards High-Quality Test Suite Generation with ML-Based Boundary Value Analysis," In 2023 10th International Conference on Dependable Systems and Their Applications (DSA), IEEE, 2023, pp. 75-85.