Semantic Approach for Increasing Test Case Coverage in Automated Grading of Programming Exercise

M. Rifky I. Bariansyah
School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Bandung, Indonesia
Email: 13517081@std.stei.itb.ac.id

Satrio Adi Rukmono
School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Bandung, Indonesia
Email: sar@itb.ac.id

Riza Satria Perdana
School of Electrical Engineering and Informatics, Institut Teknologi Bandung
Bandung, Indonesia
Email: riza@informatika.org

***Abstract*—The widely popular approach for automatic grading in computer science is to run black-box testing against the student's implementation. This kind of autograder evaluate programs solely based on their outputs given a set of inputs. However, manually writing a set of test cases with high coverage is laborious and inefficient. Hence, we explore another alternative approach in building test cases, specifically white-box testing. In theory, by knowing the internal workings of implementation, we can evaluate all possible execution paths, producing better test cases coverage, ultimately producing a complete grading. In this paper, we present research on using semantic analysis to generate test cases to determine the correctness of a student's implementation. Instead of writing test cases, the evaluator will write a reference code, a correct implementation based on the programming problem specification. We implement a system that records execution paths, detects path deviation, and checks path equivalence to analyze the semantic difference of the reference code and student's implementation. The system is built utilizing a concolic execution method for exploration and an SMT solver to solve formulas. Our experiments reveal that it is possible to automatically generate test cases and grade programming assignments by analyzing the semantic difference between reference and student implementation. Compared with grading using a random test case generator, it is evident that the system can provide better test case coverage for automatic grading in many occurrences.**

Index Terms*—automatic grading, test case generation, symbolic execution

I. Introduction

In computer science, programming exercise is used by students as a medium to implement theoretical knowledge into a program. Students rely on programming assignments grade as a study guide and feedback on their progress. However, manually grading programming assignments is time-consuming and not feasible for a large class. The more students in a class, the higher the possibility of error in grading. This problem has pushed the research effort on automatic grading. With automatic grading, students can receive feedback quickly, which increases the possibility to rework an incorrect implementation. The majority of automatic grading systems uses the black-box testing approach [1]. In this approach, the instructor or evaluator writes a set of test cases for the programming problem. The correctness of a student's implementation will then be determined using this set of test cases. However, writing a complete set of test cases, covering most if not all edge cases, requires a high amount of effort. This issue risks grading with an incomplete set of test cases, producing grades that do not reflect a student's abilities well.

This problem calls out the necessity for a different approach to writing test cases for programming exercises. This paper explores the potential of utilising a white-box testing technique, specifically semantic difference analysis, for generating test cases with better coverage. We present PyAssesment, a reference implementation of the automated grading system based on concolic execution for Python programming assignments. PyAssesment receives a reference code, i.e., a solution from the evaluator, and a student implementation as inputs. The system observes the semantical difference between the two implementations to generate a set of test cases to determine the correctness of the student implementation.

This paper is structured as follows. We first discuss the foundational basis of our work in Section II. Then, we explain our approach in generating test cases in Section III. Next, we present the result of our experiments in Section IV and discuss the key insights in Section V. Finally, we conclude and suggest further research direction in Section VI.

II. Foundational Basis

A. Automatic Grading

An automatic grading system is used for grading programming assignments in scientific computing [2]. It is built to increase the speed and capacity for evaluating students' submissions. A study shows that automatic grading on an introductory computing course positively impacts students' learning process as a feedback system. It increases the number of resubmissions, which indicates the usage of feedback to correct their implementations. In general, there are two approaches for automatic grading systems: black-box and white-box testing.

Black-box testing or functional testing utilises test cases written based on the program's specifications. This kind of testing is conducted without having access to the program's internal mechanism. A review of the current automatic grading system shows that assessing the functionality of students' code is still the most often used criteria to grade programs [1]. This grading system uses industrial testing tools such as XUnit, acceptance testing framework, web testing frameworks, or various specialised solutions such as output comparison and scripting.

White-box or structural testing utilises test cases written based on the program's source code instead of its specifications. This testing can only be conducted with access to the program's internal mechanism. While it is getting more commonly used in industry, it has found relatively rare and experimental use in automated grading systems [1].

B. Symbolic and Concolic Execution

Symbolic execution is a program analysis technique that involves executing a program. However, instead of supplying actual inputs to a program, like numbers or strings, we supply *symbols* representing arbitrary values. The goal of this method is to find possible execution paths in a program. The code snippet in Fig. 1 will be used to illustrate how symbolic execution works. First, we will supply x_0 and y_0 as the symbolic inputs for parameters x and y, respectively. Then we will execute the programs normally. Each encounter with a control flow statement will store both true and false conditions of the statement to the path constraint. The full representation of all possible execution paths can be observed in Fig. 2, with the leaves representing the path constraints.

```c
void foo(int x, int y) {
    int z = 2 * x;
    int k = 3;
    if (z > k) {
        if (y < z) {
            exit(EXIT_FAILURE);
        } else {
            assert(y != z);
        }
    }
}
```

Fig. 1: A code snippet in C [3].

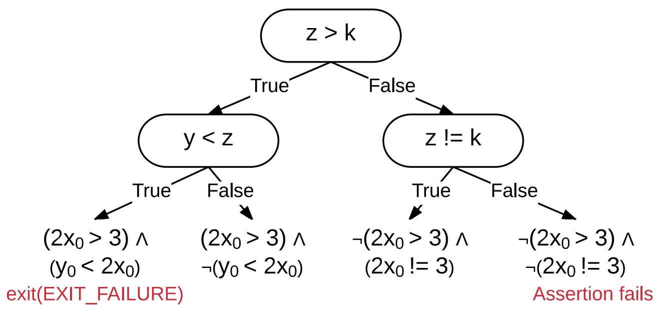

Fig. 2: Execution tree for the snippet in Fig. 1 [3].

Symbolic execution has found several uses in comparing two program implementations. Brumley et al. [4] proposed *path deviation* to find errors in different protocol implementations. Zhang et al. [5] proposed *path equivalence* to build a program logic-based approach to software plagiarism detection. Path deviation exists when given an input; two implementations will each follow a particular path. When given a different input, one implementation follows the same path as before while the other follows a different path. Path equivalence is used to ensure that the path deviations we find are actual semantics deviations rather than mere code obfuscation. These researches show the power of symbolic execution for exploring paths in program implementation.

Concolic execution or *dynamic symbolic execution* is a variation of symbolic execution run concurrently with concrete execution [6]. Instead of only keeping symbolic variables, it will also store an actual value for the variables. With this approach, complex constraints that a constraint solver cannot solve can be simplified by replacing symbolic variables with concrete values. The path constraint will be used incrementally to explore execution paths, delivering a higher coverage.

This method can be used to capture a program's semantics. Semantics describes what a program means. It is a rule of interpreting syntax that has no direct meaning [7]. Some examples of research related to semantic difference analysis follow.

a) Semantic Diff: Semantic Diff is a tool for summarizing the effects of modification. The tool takes two versions of a procedure and generates a report summarizing the semantic differences between them [2].

b) Symdiff: Symdiff is a language-agnostic semantic difference tool for imperative programs. This tool performs equivalence checking and displays semantic differences [8].

III. Test Cases Generation

A. Basic Idea

The basic idea of the system is to explore and observe semantic differences between a reference and a student implementation to generate a complete set of test cases for grading. For the exploration, we use concolic execution, a higher coverage variant of symbolic execution. After exploration, we detect path deviation [4] and path equivalence [5] between the paths. We seek path deviation and equivalence to explore both implementations completely and find paths that are semantically different. Test cases generated from semantically different paths might lead to different outputs in two implementations that point out incorrectness in them.

B. Design

The flow diagram for our design is illustrated in Fig. 3. The process can be divided into three phases: concolic exploration, path deviation detection, and path equivalence checking. The main goal of the research is to generate better test cases, but we also designed the system to include the grading process using the generated test cases. Firstly, the system generates an early set of test cases with concolic exploration. Then, we supply each iteration with this early set. While the set is not empty, the grading process continues. We pick a test case in each iteration and execute both the student and reference implementation against this test case. If the outputs between the implementations are different for a test run, the student implementation will be considered wrong for that test case. If the outputs are equal, the process will continue to path deviation checking. If there is no path deviation detected, then the process will skip to the next iteration. Otherwise, the process will continue to path equivalence checking. Finally, if there is a path equivalence, the grading will continue to the next iteration; otherwise, it will consider that the student implementation is wrong for that test case. To sum up, the set of test cases might come from concolic exploration, path deviation detection, and path equivalence checking.

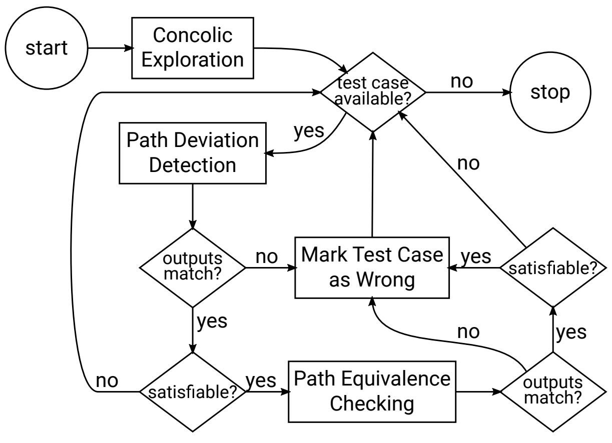

Fig. 3: The flow diagram for our grading system.

a) Concolic Explorer: The concolic explorer is illustrated in Fig. 4. The component will explore reference and student implementation for possible execution paths using concolic execution, hence the name concolic explorer. Each execution path from the exploration will be paired with a test case. The test cases will be compiled as an output of this component.

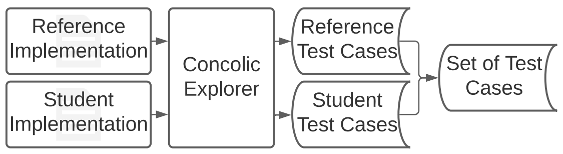

Fig. 4: The concolic explorer.

b) Execution Component: The purpose of this component is to execute reference and student implementation with the same input. The input comes from the concolic explorer, counterexample from path deviation detection, or counterexample from path equation checking. Every time a truth value testing is performed (e.g., conditional statements or loop conditions), the statement will be appended to the path constraint. Thus, the component will return output and path constraint from both implementations as the execution result.

c) Path Deviation Detection: The path deviation detection is illustrated in Fig. 5. This component aims to determine if there is any path deviation between reference and student execution paths. This component takes reference and student path constraints as inputs. The process can be divided into two phases: formula building and constraint solving. Given two constraints, an input that shows path deviation will satisfy one path constraint but not the other.

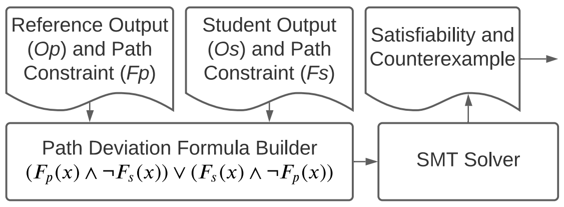
Reference Output (Op) and Path Constraint (Fp)
Student Output (Os) and Path Constraint (Fs)
Satisfiability and Counterexample
Path Deviation Formula Builder
$(F_p(x) \wedge \neg F_s(x)) \vee (F_s(x) \wedge \neg F_p(x))$
SMT Solver

Fig. 5: Path deviation detection.

To determine if such input exists, we check the satisfiability of (1), as used by Brumley et al. [4].

$$(F_p(x) \wedge \neg F_s(x)) \vee (F_s(x) \wedge \neg F_p(x)) \tag{1}$$

$F_p(x)$ is the reference path constraint, and $F_s(x)$ is the student path constraint. In general, if the formula is not satisfiable, no path deviation is detected, and the grading will skip to the next iteration. However, if the formula is satisfiable, there is a path deviation detected. The counterexample is the input that points to a path deviation.

d) Path Equivalence Checking: The path equivalence checking is illustrated in Fig. 6. This component aims to determine if there is any path equivalence between reference and student execution paths. We do this to make sure the path deviation is accurate and not caused by obfuscation. This component takes reference and student output and path constraint as inputs. Similar to path deviation detection, the process can be divided into formula building and constraint solving. For example, given two constraints, an input that shows a lack of path equivalence will satisfy the conjunction of the path constraints and inequality of the outputs.

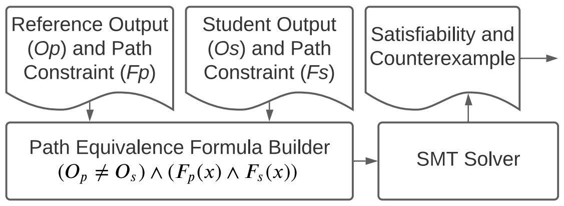
Reference Output (Op) and Path Constraint (Fp)
Student Output (Os) and Path Constraint (Fs)
Satisfiability and Counterexample
Path Equivalence Formula Builder
$(O_p \neq O_s) \wedge (F_p(x) \wedge F_s(x))$
SMT Solver

Fig. 6: Path equivalence checking.

To determine if such input exists, we check the satisfiability of (2), as used by Zhang et al. [5].

$$(O_p \neq O_s) \wedge (F_p(x) \wedge F_s(x)) \tag{2}$$

O_p is the reference output, and O_s is the student output. In general, if the formula is not satisfiable, the paths are considered equivalent, and the grading will continue to the next iteration. If the formula is satisfiable, there is a lack of path equivalence. The counterexample is the input that shows incorrectness in the student implementation.

IV. Experiments

We evaluate the system using problems and student submissions from freshman-level programming courses for the informatics major in our institution. For the experiments, we wrote a reference implementation for each problem. We also grade the student implementations using a random test case generator to compare against PyAssesment.

A. Reference Implementation

The reference implementation "PyAssesment" is written in Python for grading Python programs. We leverage PyExZ3 for concolic exploration early in the grading process [9]. PyExZ3 is a concolic execution tool (or dynamic symbolic execution referenced on the paper) for Python programs. PyExZ3 also inspires the technique used in the execution component to store path constraints. To solve constraint problems, we use Z3 as the SMT solver [10]. Z3 is a prominent SMT solver widely used in various software verification and analysis applications. Z3 is used in PyExZ3, path deviation detection, and path equivalence checking. The current version of PyAssesment has two limitations: it can only grade implementations in the form of function with a return statement, and it can only grade implementations with integer inputs.

B. Studied Cases

We studied six programming exercises for freshman-level programming courses in an Informatics bachelor program. We use the following exercises: 1) max_3, finding maximum value out of three integers, 2) $square$, building a square of string from characters '*' and '#', 3) $student\_grade$, returning index ('A'–'E') based on grade (0–100), 4) $is\_allowed\_to\_buy$, checking whether a grocery purchase is allowed based on a fixed set of rules, 5) $no\_of\_triangle$, returning the number of triangles that can be created out of three lengths, 6) $water$, returning water form based on temperature. We examine ten different submissions for each exercise, i.e., no two submissions use the same approach. For each exercise, we wrote a reference implementation to be used in grading. Because of the implementation limitations, we translate the submissions into a function that takes integer parameters and has a "return" statement.

C. Running Example

We use max_3 to demonstrate how the system works. Fig. 7 shows our reference code (max_3_ref) and one student's implementation (max_3_std). First, an early set of test cases is generated with concolic exploration. From the set, we take one test case, for example ($a = 0$, $b = 0$, $c = 1$), to be executed using both implementations. The execution will return 1 in both implementations and path constraints, as shown in Table I. Then, the system detects if there is any path deviation between the two paths. A path deviation formula is built from the path constraints, also shown in Table I. The path deviation formula is proven to be satisfiable by the SMT solver, indicating a path deviation. A new test case is generated from the SMT solver, ($a = 0$, $b = 0$, $c = 0$), and is executed by both implementations. The execution returns 1 in both implementations and path constraints. The system makes sure the path deviation it finds is a semantic deviation by checking path equivalence; the formula is shown in Table II. The path equivalence formula is proven to be satisfiable, indicating the lack of path equivalence. The SMT solver returns ($a = 0$, $b = 0$, $c = -1$) as counterexample. With the new input, the reference code returns 0 while the student's implementation returns -1, signifying a detected mistake. The final set of test cases generated is used to grade the student implementation, resulting in a 91.6% score.

TABLE I. Running Example Part 1.

Path Constraints from 1st Execution	
Reference	And(a >= b, a < c, b >= a, b < c)
Student	And(a <= b, b <= a)
Path Deviation Formula in Z3 Expression	
Or(And(And(a >= b, a < c, b >= a, b < c), Not(And(a <= b, b <= a))), And(And(a <= b, b <= a), Not(And(a >= b, a < c, b >= a, b < c))))	

TABLE II. Running Example Part 2.

Path Constraints from 2nd Execution	
Reference	And(a >= b, a >= c)
Student	And(a <= b, b <= a)
Path Equivalence Formula in Z3 Expression	
And(a ! = c, And(And(a >= b, a >= c), And(a <= b, b <= a)))	

max_3_ref

```
def max_3_ref(a, b, c):
  if a >= b and a >= c:
    return a
  elif b >= a and b >= c:
    return b
  else:
    return c
```

max_3_std

```
def max_3_std(a, b, c):
  if a > b and a > c:
    return a
  elif b > a and b > c:
    return b
  else:
    return c
```

Fig. 7: Two implementations of max_3.

D. Result: Grading Scores

We present the score differences between grading with randomly generated test cases and PyAssesment's test cases. The random test cases are generated using a pseudo-random Python module. The grading result is shown in Fig. 8. The y-axis represents the score for each student implementation. Blue bars represent grading results with random test cases and red bars for PyAssesment's test cases. In general, the scores

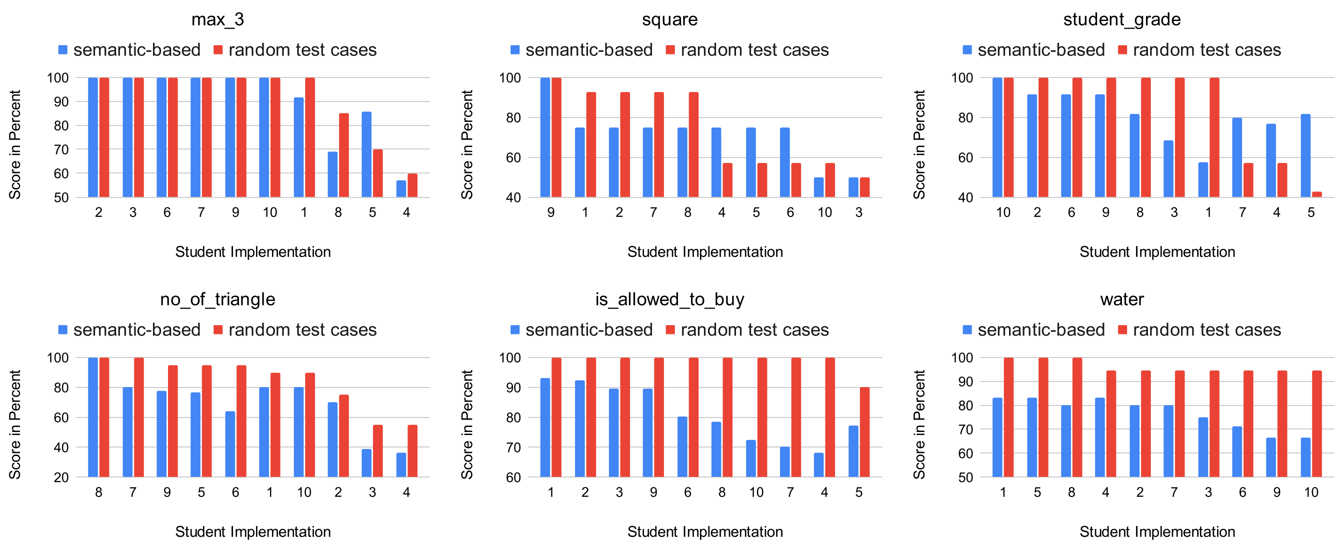

Fig. 8: Grading Result.

are higher when using randomly generated test cases. These scores indicate failing cases discovered by PyAssessment but not covered by the randomly generated test cases.

E. Result: Branch Coverage

We also analyze the branch coverage differences between randomly generated test cases and PyAssesment's test cases. The grading result is depicted in Fig. 9. The y-axis represents the branch coverage for each student implementation. Blue bars represent random test cases coverage and red bars for PyAssesment's test cases coverage. Overall, the branch coverages are higher for PyAssesment's test cases. All problems except $no\_of\_triangle$ and $water$ are 100% covered by PyAssesment. The coverages from random test cases are also quite high in more straightforward problems, like $square$ and max_3. However, as we discuss later, branch coverage is not a good indicator for grading completeness.

TABLE III. Number of Passes in $student\_grade$.

Condition	PyAssesment's Test Cases	Random Test Cases
a < 0	2 (15.3%)	6 (42.8%)
a > 100	1 (7.6%)	0 (0%)
80 ≤ a ≤ 100	2 (15.3%)	1 (7.1%)
73 ≤ a ≤ 79	2 (15.3%)	0 (0%)
65 ≤ a ≤ 72	1 (7.6%)	1 (7.1%)
57 ≤ a ≤ 64	2 (15.3%)	4 (28.4%)
50 ≤ a ≤ 56	1 (7.6%)	1 (7.1%)
35 ≤ a ≤ 49	1 (7.6%)	1 (7.1%)
0 ≤ a ≤ 35	1 (7.6%)	0 (0%)

V. Discussion

The higher scores in grading using randomly generated test cases indicate higher coverage when grading with semantically generated test cases. The difference is especially noticeable in complex problems like $is\_allowed\_to\_buy$ that involves many complex statements. PyAssesment is designed to find a condition specifically from the statements it encounters during execution, which random test cases are less likely to find. In grading our sample student implementation, random test cases deliver no wrong cases as the final result. Meanwhile, as we see in the running example, the semantic-based test case generation produced ($a = 0$, $b = 0$, $c = -1$) from path equivalence detection, which returns different output in the student implementation. The student's work is only wrong when condition $a = b$, $a > c$, and $a > b$ is satisfied, which is not bound to be found with random test cases.

Nevertheless, as we can see, some students achieve higher scores in some exercises when graded using PyAssesment's test cases. We found that this happens because many test cases from a random generation point to the same path. This phenomenon can be best explained by observing the grading of the problem $student\_grade$, where PyAssesment's test cases return a score of 76.8% and random test cases return 57.1%. Table III shows the number of times a condition in $student\_grade$ is passed. The student implementation is proved wrong when the input is less than 0 or greater than 100. Six of the 20 random test cases pass the condition where the input is less than 0, while none passes the condition where the input is greater than 100. At the same time, PyAssesment's set of test cases passes both wrong conditions. Hence, in this case, the lower score using the random test cases does not show a higher test case coverage.

We also analyze the branch coverage of those test cases to see if it is a good indicator for grading completeness. Complex problems manifest higher branch coverage in PyAssesment test cases very well, while random test cases can only achieve high branch coverage in simple problems. Although, some PyAssesment test cases have low branch coverage when there is unreachable code in the implementation, as encountered in the $no\_of\_triangle$ case (Fig. 9). Ultimately, we find that

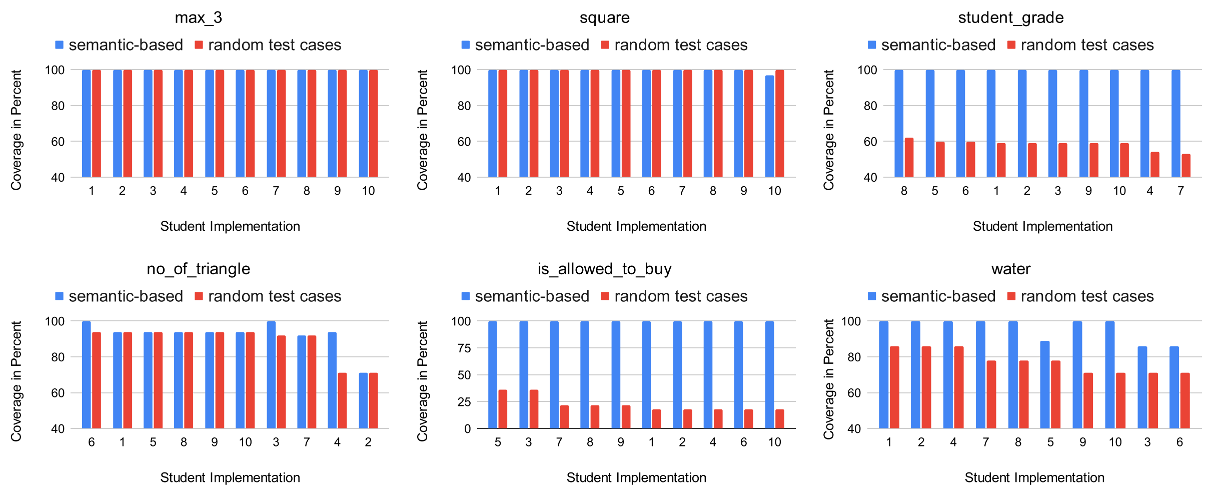

Fig. 9: Branch Coverage.

100% branch coverage does not indicate a complete grading. In grading max_3, both PyAssesment's and random test cases return a 100% branch coverage. Nevertheless, PyAssessment test cases expose a mistake in the student implementation, but random test cases do not. Investigation on the branch coverage shows that it is possible to pass all branches without encountering a wrong condition. In this example, we need to have a test case that satisfies $a = b$, $a > c$, and $a > b$ to point the mistake in student implementation. In that condition, our sample student implementation will go to the latest *else* statement and return the variable c. To pass that particular branch, we can also use an input of ($a = 0$, $b = 0$, $c = 1$), which does not satisfy $a = b$, $a > c$, and $a > b$. This example shows that it is possible to have 100% branch coverage and incomplete grading simultaneously. Therefore, branch coverage is not the best indicator for grading completeness. Based on the result, testing using another type of coverage might be a good idea for future work. One type that might fit this problem is condition coverage, which has not yet been explored for Python at the time of writing.

VI. Conclusion

We have proposed a semantic-based test case generation. Semantic difference is analyzed by running concolic exploration, finding path deviation, and checking path equivalence. By utilizing this method, we can provide test cases that are automatically generated and more complete than randomly generated test cases. Semantic-based test cases also provide higher branch coverage, but branch coverage is not a good grading completeness indicator.

Future Work

Another approach to grading student assignments is using the semantic difference directly instead of for generating test cases. With this model, the system can produce a score based on how semantically similar are the student and reference implementation. In contrast with the PyAssignment, which ultimately executes a black-box grading against test cases generated using a white-box technique, this approach will be using a white-box method for grading.

Acknowledgment

This paper is a part of more extensive work in automatic grading involving our peers Kevin Sendjaja, Irfan Sofyana Putra, and Bram Musuko Panjaitan.

References

[1] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments," in *Proceedings of the 10th Koli calling international conference on computing education research*, 2010, pp. 86–93.

[2] D. Jackson, D. A. Ladd *et al.*, "Semantic diff: A tool for summarizing the effects of modifications." in *ICSM*, vol. 94, 1994, pp. 243–252.

[3] J. Son, "Symbolic Execution – CS261 Security in Computer Systems (Scribe Note)," https://inst.eecs.berkeley.edu/~cs261/fa17/scribe/SymbolicExecution.pdf, 2017, accessed: 2021-07-23.

[4] D. Brumley, J. Caballero, Z. Liang, J. Newsome, and D. Song, "Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation." in *USENIX Security Symposium*, 2007, p. 15.

[5] F. Zhang, D. Wu, P. Liu, and S. Zhu, "Program logic based software plagiarism detection," in *2014 IEEE 25th international symposium on software reliability engineering*. IEEE, 2014, pp. 66–77.

[6] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[7] J. Euzenat, P. Shvaiko *et al.*, *Ontology matching*. Springer, 2007, vol. 18.

[8] S. K. Lahiri, C. Hawblitzel, M. Kawaguchi, and H. Rebêlo, "Symdiff: A language-agnostic semantic diff tool for imperative programs," in *International Conference on Computer Aided Verification*. Springer, 2012, pp. 712–717.

[9] M. Irlbeck *et al.*, "Deconstructing dynamic symbolic execution," *Dependable Software Systems Engineering*, vol. 40, p. 26, 2015.

[10] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.