Delft University of Technology

Artificial Intelligence for Software Testing and Reverse Engineering

CS4110

Group 22: Final Lab

Authors:
B. Han (4389336), S.X. Zhang (4660129)

April 17, 2022

TASK 1

Implementations for lab 1,2, and 3 can be found on the following GitHub repository.

Lab 1 - Fuzzing

1.1 A description of the used search technique including pseudo-code.

Fuzz testing is an automated software testing technique that involves generating random invalid and unexpected data as inputs for a particular computer program. For this assignment we used Reachability problems from the RERS 2020 problems. For the reachability problem the fuzzer needs to generate a test case. In our case the fuzzer generates a "Trace" consisting of possible input symbols for the program. The objective of the fuzzer is to reach the max No. unique branches visited and trigger as much errors in a specific problem. For this task we implemented two types of fuzzers.

- **Random Fuzzer:** This fuzzer is completely random and simulates a brute force technique. (Since we think the code for the Random Fuzzer is trivial we moved the pseudo-code to the Appendix)
- **Hill Climbing Fuzzer:** Hill climbing is a local search algorithm and starts with a random Trace and improves it until an optimal is found. In our case we want to maximize the branches visited and minimize the sum of distances. Thus, for each trace we compare by normalizing over the sum of distances of each branch, D, and over the No. unique branches visited, N:

$$NORMALIZATION(D, N) = \frac{D}{D + N} \tag{1}$$

  Based on the best trace, we construct 20 permutations. Permutations consist of adding and swapping one and/or two input symbols. We again pick the best permutation and run the cycle again with this new trace and construct 20 more permutations. If no better trace/permutation is found we start again with a random trace.

Algorithm 1 Hill Climbing Fuzzer

```
BetterTraceFound = False
while Time limit not reached do
    if BetterTraceFound then
        input_traces ← Create 20 permutations of the best trace found in previous round
    else
        input_traces ← 20 random traces
    end if
    for each_trace_in_input_traces do
        while Encountered_new_branch do
            sumDistances += compute current branch distance
            Count No. of visited unique branches and errors of the input trace
            Count Total No. visited brachces and errors found
        end while
        Compare input traces with best input trace found so far (pick min. NORMALIZATION)
        if Better trace updated then
            BetterTraceFound ← True
            Save the best trace for next round (Permutation)
        else
            Else BetterTraceFound ← False
        end if
    end for
end while
Log Total No. visited branches visited and errors found
```

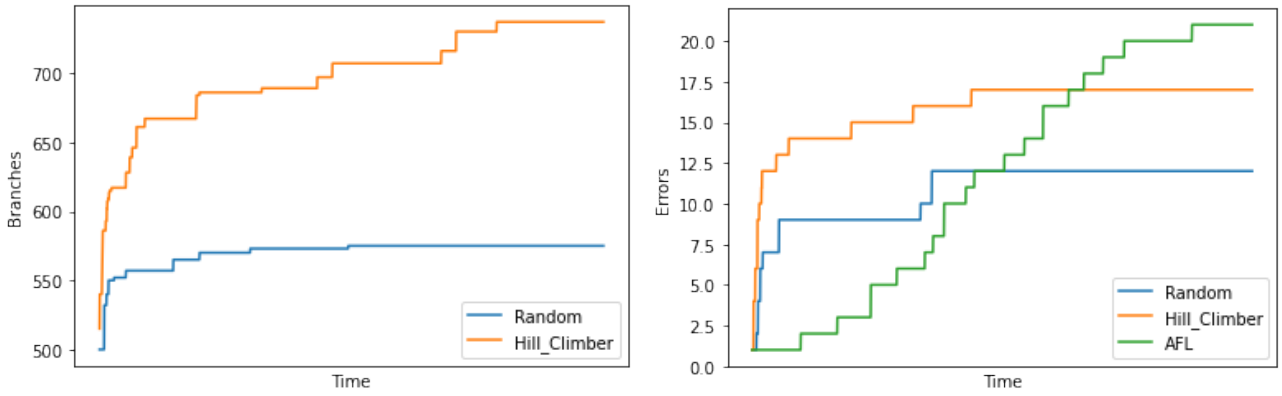

Figure 1: Problem 17 plot for branches (left) and crashes over time (right)

1.2 A run of your solution run on one of the RERS problems and explain the results it gives.

Both fuzzers were run on Problem 17. We compare the unique branches visisted and errors found over a time period of 15 minutes (see table 1). Results are given in the following tables and figures. As apparent from the figures the Hill Climbing Fuzzer is able to reach more branches and errors the Random Fuzzer.

Problem 17	**Random F.**	**Hill Climbing F.**
Unique Branches	573	**737**
No. Errors	12	**17**
Error Codes	[96, 72, 23, 78, 11, 99, 53, 97, 69, 25, 35, 46]	[96, 72, 61, 23, 12, 78, 66, 11, 99, 53, 97,69, 25, 35, 46, 0, 2]

Table 1: Branch and error analysis problem 17

We have also included the state-of-the-art American Fuzzy Lop (AFL) tool for comparison purposes. Although our Hill Climbing Fuzzer is able to find more reach more branches and errors than the Random Fuzzer, it does not reach the same results as AFL. A reason might be that Fuzzer ends up in a local optima and better traces are not explored. Also, permutation strategy might still be too random.

Based on Figure on the right we can see that our hill climbing fuzzer is able to find more errors in the beginning, resulting in a logarithmic graph, whereas AFL finds errors in a linear manner and is able to find more errors over time. This behavior can be explained by how we implemented a lot of randomness at the start. Furthermore, randomness is again applied when no better trace is found. The AFL might be investigating deeper on specific traces instead of resorting to random traces.

1.3 Two strengths and two weaknesses of each method.

1. Strengths
   (a) Fuzzing with randomness can easily create many inputs and thus explores a large input space of test cases. Randomness can even trigger unexpected behaviour the developer/tester did not account for.
   (b) Fuzzing is an almost completely automated testing technology that drastically reduces the manual effort for developers/testers.
2. Weaknesses
   (a) Since the Hill Climbing is a local search algorithm one of its weaknesses is that it can converge to a local optimum. instead of finding the global optimum.
   (b) It is difficult to cover specific cases. Since we are trying to reach the max No. branches visited we should try and trigger both True and False branches of an If-statement. Since our Hill Climbing utilizes permutations that are still quite "random" it is hard to control what should be triggered.
   (c) Many inputs can cover the same paths through a program.

Lab 2 - Symbolic Execution

2.1 A description of the used search technique including pseudo-code.

Symbolic execution is used to explore as many program paths as possible resulting in high-coverage test suites.. For each path it generates a set of concrete input values. In our case it generates satisfiable traces to be run on the RERS problem 2020.

First our algorithm starts generating a random trace. With every new branch encountered the algorithm starts solving the *opposite_branch* and generates satisfiables inputs, which are stored in a SATISFIABLES LinkedList awaiting to be executed in next iterations. Unsatisfiables inputs are stored in UNSATISFIABLES s.t. we can ignore these in next iterations for performance reasons. We extend satisfiable traces to have a length of 25 to ensure that it explores a bit more in a specific path. Traces found with length greater than 25 will not be extended. HashSets are used to keep track of newly VISITED_BRANCHES and FOUND_ERRORS. If we run out of satisfiables inputs to run (SATISFIABLES is empty) we start again with a random trace.

Finding satisfiable input requires solving Satisfiability modulo theories (SMT). Luckily, MicroSoft provides us with the Z3 solver which is an automatic theorem solver and determines whether certain inputs are satisfiable or unsatisfiable for a specific program.

Algorithm 2 Symbolic Execution

```
1: while Time limit not reached do
2:     if SATISFIABLES not empty then
3:         for each_trace_in_SATISFIABLES do
4:             while Encountered_new_branch do
5:                 if Current branch_line_nr + boolean_value already visited then
6:                     Skip this encountered branch
7:                 end if
8:                 Create Expression for executing opposite_branch
9:                 if UNSATISFIABLES does not contain opposite_branch then
10:                    Start solving opposite branch using Z3 solver
11:                    SATISFIABLES ← Found satisfiables trace inputs (extended if length < 25)
12:                    UNSATISFIABLES ← Found unsatisfiable trace inputs
13:                else
14:                    Skip unsatisfiable trace input for performance reasons
15:                end if
16:                VISITED_BRANCHES ← current branch_line_nr + boolean_value
17:            end while
18:            ERRORS_FOUND ← Error outputs
19:        end for
20:    else
21:        Run random trace
22:    end if
23: end while
24: Log Total No. VISITED_BRANCHES and ERRORS_FOUND
```

2.2 A run of your solution run on one of the RERS problems and explain the results it gives.

We start analyzing problem 17 again and also want to compare it against the Hill Climbing Fuzzer. The symbolic Execution Algorithm also ran for 15 minutes to keep execution time of both strategies consistent. (*Note: since in our first lab assignment we did not specify branch ID with their respective boolean value and only used the line number, the No. visited branches for the Symbolic execution will be significantly higher.*)

Problem 17	**Hill Climbing F.**	**Symbolic Execution**
Unique Branches	737	**4443**
No. Errors	17	**30**

Table 2: Branch and error analysis Problem 17

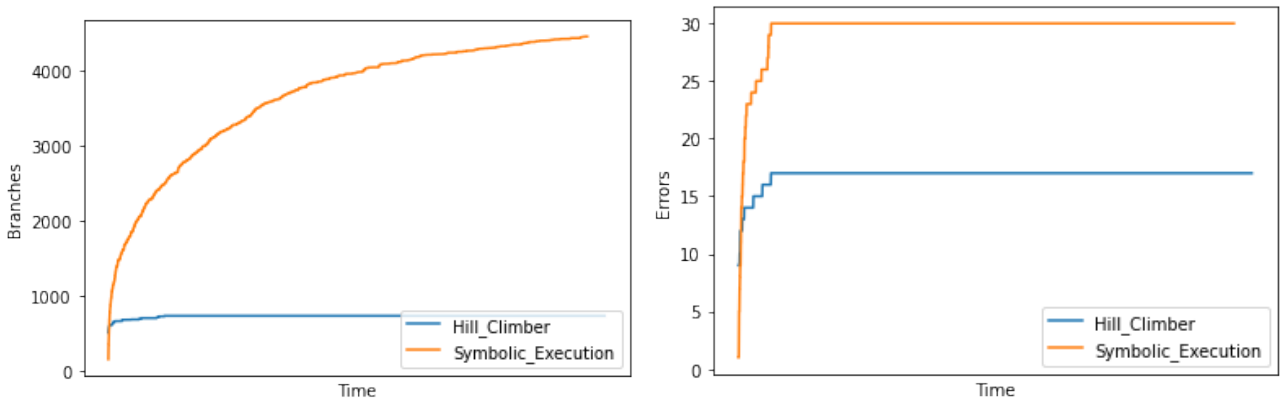

Figure 2: Problem 17 plot for branches (left) and crashes over time (right)

Based on the numbers in table 2 and the convergence graphs (see figure 2) we argue that Symbolic execution performs better than our hill climber fuzzer. Even though the No. visited branches of the Hill Climbers Fuzzer does not show correct results, we can still make the claim that Symbolic Execution has reached a significant high amount of unique branches. Where symbolic execution tries to trigger the specific branches and is more controlled than the hill climber which uses random permutations. The logarithmic curve also indicates that Symbolic Execution has good control on how specific branches are triggered.

2.3 Two strengths and two weaknesses of each method.

1. Strengths
   (a) Symbolic execution allows us to execute a program through all possible execution paths, thus achieving all possible path conditions. In terms of branch coverage Symbolic Execution has more control over what branches it should trigger. We take full advantage of the Z3 solver to find satisfiables input traces and different paths of the program.
   (b) The ability to generate concrete test inputs. From a test generation perspective, it allows the creation of high-coverage test suites, while from a bug-finding perspective, it provides developers with a concrete input that triggers the bug.
2. Weaknesses
   (a) Key disadvantage of classical symbolic execution is that it cannot generate an input if the symbolic path constraint along a feasible execution path contains formulas that cannot be (efficiently) solved by a constraint solver (for example, nonlinear constraints).
   (b) Path Explosion: Symbolically executing all feasible program paths does not scale to large programs.

Lab 3 - Automatic Code Patching

3.1 A description of the used search technique including pseudo-code.

Genetic algorithms is inspired by Charles Darwin's theory of natural evolution, representing the process of natural selection where the fittest individuals are selected for reproduction in order to produce the optimal offspring for the next generation. The steps of genetic algorithms are:

1. Natural selection: individuals that best fit the natural environment survive and the worst die.
2. Reproduction: survived individuals generate offspring.
3. Mutation: offspring inherits properties of its parents, with some mutations.
4. Iteration: generation by generation the new offspring fits better the environment than its ancestor.

In this assignment, fault localization using the Tarantula score was used to determine which operators are more likely to be faulty by computing how frequently an operator is used in a failing test.

In our algorithm, the first operation is creating an initial population which are copies of the the OperatorsTracker.operators. While the program has not reached its time limit yet, the Tarantula scores are calculated and logged for each individual in the population. When the population is empty, the reproduction starts. The first step is sorting the individuals on fitness. Then crossover is applied on the #BEST_INDIVIDUALS, which is a set number defined before running the algorithm. In crossover, a random point is selected to cut and the operators of two individuals. Following crossover, mutation is applied on the #BEST_INDIVIDUALS. For each individual roulette selection is used to select an operation to mutate. For each operation, a random different operator is picked to replace it. The roulette selection uses the tarantula score to assign each operator a probability. The higher the tarantula score, the higher the probability, the more chance the operator is selected to be mutated.

See APPENDIX A: Algorithm 4 for Genetic Algorithm pseudo code

3.2 A run of your solution run on one of the RERS problems and explain the results it gives.

To experiment with different mutation rates we provide the following Hyperparameters and two different settings for comparing mutation rates (See table 3).

- POPULATION: The number of individuals in a population.
- BEST_INDIVIDUALS: best n-individuals to get to the next generation without picking.
- CROSS_OVER TYPE: 1-point crossover
- GENE_MUTATION_SIZE: Number of genes that need to be mutated.

Experiment	**setting 1**	**setting 2**
POPULATION	10	20
BEST_INDIVIDUALS	3	8
CROSS_OVER_TYPE	1-point	1-point
GENE_MUTATION_SIZE	2	6

Table 3: Hyperparameter settings for experiments

Setting 2 considers a bigger population, takes more best individuals into account for reproduction, and does more mutations on an individual. Both settings ran for 20 minutes. We can see that Setting 2 runs fewer generations than Setting 1 (See figures 3). This can be explained due to Setting 2 having a bigger population. As Setting 2 has a bigger population, crossover and mutation take longer. Furthermore, more mutations are applied in Setting 2, which also increases the runtime. However, we can conclude that Setting 2 is able to reach a higher fitness score in the same amount of time. Setting 2 provides better parameters for mutations to reach deeper in the program.

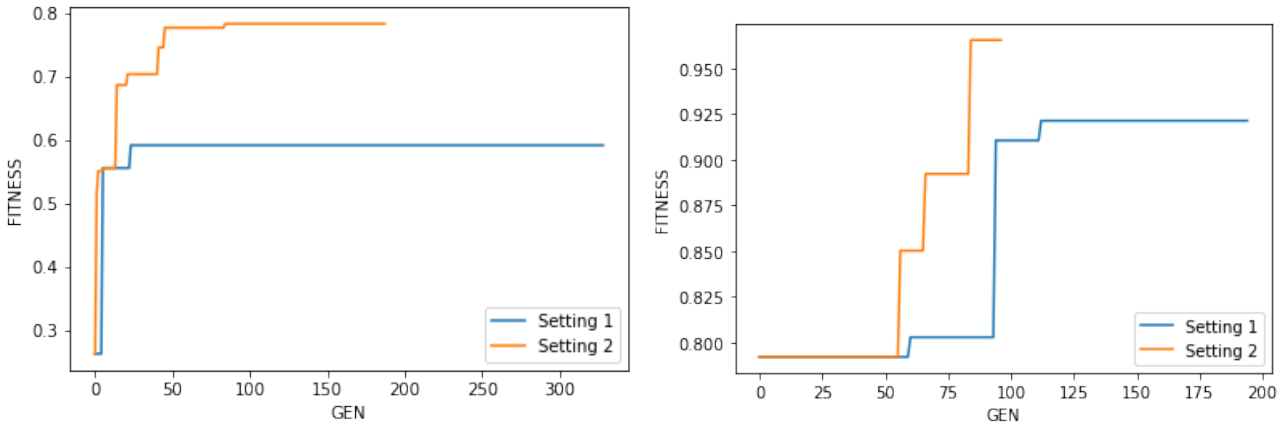

Figure 3: Convergence graphs problem 12 (left) and problem 13 (right) settings 1 vs. setting 2

3.3 Two strengths and two weaknesses of each method.

1. Strengths
   (a) Genetic algorithms stochastically selects the individuals based on their fitness to perform crossover and mutation on. By taking the best individuals and altering these it is most likely to find even fitter individuals.
   (b) Parallelism, as the algorithm can easily be modified to other problems. Different types of crossover, such as one-point or two-point can be used and different types of mutations can be implemented.
2. Weaknesses
   (a) The algorithm may be able to find even fitter individuals when performing alterations on individuals that did not fall under the best individuals.
   (b) The success of the algorithm is highly dependent on the fitness function, mutation and crossover rate and population size. A bad fitness function may lead to problems such as unable to find the solution to a problem or returning a wrong solution. A population size that is too small, will not give the algorithm enough space to produce accurate results. A high mutation and crossover frequency may change the individual too much to ever converge.

Task 2

Project Selection

For selecting projects for analyzing generated EvoSuite test cases we try to keep the projects simple s.t. classes are not to complex and we can focus more on the analysis of generated test cases and seeding faults. For every project there is a corresponding GitHub Repository that includes thes source code, generated test cases, and seeded faults. The repository includes the folders $faultA$ and $faultB$. FaultA includes code where faults were seeded that do not trigger a test case. FaultB includes code where faults do trigger a test case.

Project 1: Exam class for grading based on points

The first project was chosen to be very simple and was used more as a tutorial to get us familiarized with EvoSuite and its workings. The full project and generated tests can be found on the Project 1 GitHub Repository.

The project contains one class, Exam, and consists of one function: grade(Boolean cheated, Int points). It consists of 7-if statements and Based on the two parameters it will output a grade ranging from 1-5. EvoSuite successfully generated tests for the class and reached overall coverage of 92%.

```
Total number of classes in the project: 1
[INFO] Number of classes in the project that are testable: 1
[INFO] Number of generated test suites: 1
[INFO] Overall coverage: 0.91625
```

Test generated by EvoSuite are sensible. 12 test are generated that reach 100% branch and line coverage and 92% overall coverage (see table 4). Looking at the tests it covers all branches and edge-cases. It also tries -1 as value for points which should return an $IllegalArgumentException$.

TARGET_CLASS	criterion	Coverage	MutationScore
nl.tudelft.evo.Exam	LINE	1.0	0.70
nl.tudelft.evo.Exam	BRANCH	1.0	0.77
nl.tudelft.evo.Exam	WEAKMUTATION	1.0	0.83

Table 4: EvoSuite test suites Coverage Project 1

To seed a fault that triggers a test we changed one of the operators in the if-statement. See Appendix B for the original code (figure 4) and code with bug (figure 5). $Test00$ got triggered and we see that it covers the edge-case. It is apparent that EvoSuite generated test cases that covers all edge-cases.

For seeding a bug that does not trigger a test we tried changing every operator. For this project no seed can not trigger a test. EvoSuite generated excellent test cases and covers all the edge cases for a simple project like this. The EvoSuite tests thus cover all expected behaviour.

Project 2: GUI for simple calculator

Project 2 was chosen to be more complex than project 1. The project consists of a very basic calculator application created with Java Swing. It was cloned from the following GitHub link. In this project we wanted to see whether EvoSuite works with projects that are more GUI based. Since this project was build in Swing and was focused more on the output of a simple GUI we were curious what test cases EvoSuite will generate. To see the generated test cases see the Project 2 GitHub Repository.

EvoSuite succesfully generated test for the Calculator class. However we see that the overall coverage is 25%.

```
Total number of classes in the project: 1
[INFO] Number of classes in the project that are testable: 1
[INFO] Number of generated test suites: 1
[INFO] Overall coverage: 0.25
```

Test generated for this project were not sensible. Looking at test generated with line and branch criterion we see a coverage of only 10% (See table 5). Only one test case was generated and it involved one exception that was thrown in the project. Since the project is more GUI based it is reasonable to say that EvoSuite does not fit this project. Seeding faults seemed trivial for this specific project since it was more GUI based and EvoSuite did not generate the necessary test cases. For automated UI testing for applications build in Swing one might prefer using FEST.

TARGET_CLASS	criterion	Coverage	MutationScore
com.houarizegai.calculator.Calculator	LINE	0.11	0.0
com.houarizegai.calculator.Calculator	BRANCH	0.10	0.0
com.houarizegai.calculator.Calculator	WEAKMUTATION	0.0	0.0

Table 5: EvoSuite test suites Coverage Project 2

Project 3: Snake Game

In this project we would like to analyse a Snake Game. For this project we kept it simple w.r.t. class complexity and one that is not GUI heavy. This project is cloned from the following GitHub link. The project is a simple snake game that has a *Snake* and *Board* class. Much of the functionality is implemented in the *Board* class. For the generated test cases and faults that were seeded please refer to our Project 3 GitHub Repository.

EvoSuite generated a test suite for the Board class with 33% overall coverage. EvoSuite could not generate a test case for the Snake class for unknown reasons.

```
Total number of classes in the project: 2
[INFO] Number of classes in the project that are testable: 1
[INFO] Number of generated test suites: 1
[INFO] Overall coverage: 0.334375
```

Test generated for this project are quite sensible. Looking at test generated with line and branch criterion we see a coverage of only 68% and 42% respectively (See table 6). EvoSuite generated 5 test cases and these involve 1 test for an exception and the remainder of the tests involve more around *ActionEvents*.

Seeding faults of type A (test cases not triggered) were made in the *checkCollision*() method. For the introduced bugs please see the $faultA$ folder with the java file. The introduced bugs can be found in the comments at the top. The changes did not trigger tests generated by EvoSuite. We tried to see if EvoSuite takes care of edge cases in the collision function, in this case it does not. The reason for this might be that each if-statement changes a value in an array but does not return a value. These small changes might not have great impact on the game.

Seeding faults of type B (test cases triggered) we made changes throughout the whole java file. For the introduced bugs please see the $faultB$ folder. These bugs triggered 3 out of 5 test cases to fail. The bugs are very crucial to the program since we made changes in a for-loop and set some initial variables to different values. These bugs have big influence on the game and were caught by generated EvoSuite tests.

TARGET_CLASS	criterion	Coverage	MutationScore
com.zetcode.Board	LINE	0.68	0.03
com.zetcode.Board	BRANCH	0.42	0.03
com.zetcode.Board	WEAKMUTATION	0.56	0.04

Table 6: EvoSuite test suites Coverage Project 3

EvoSuite issues

- Minor setup issue where I had to downgrade to Java 1.8 and some environment variables that needed to be setup correctly to run the dependencies.
- Option to only check the Branch coverage instead of overall coverage generated by (mvn evosuite:generate). For extra functionalities I had to use the evosuite-1.0.6.jar file and execute (mvn dependency:copy-dependencies)
- Difficulty in finding the right Java projects to analyse.

Task 3

For the task 3, ASTOR from the docker image was used and it was run on Math-issue-280 from the examples in ASTOR.

GP

ASTOR was run four times using the GP with the following command

```
java -cp target/astor-*-jar-with-dependencies.jar fr.inria.main.evolution.AstorMain
-mode jgenprog -srcjavafolder /src/java/ -srctestfolder /src/test/
-binjavafolder /target/classes/ -bintestfolder  /target/test-classes/
-location /home/str/astor/examples/Math-issue-280/ -dependencies examples/Math-issue-280/lib
```

In every run except the fourth of ASTOR using GP, 6 patches were found. See Table 7 for the results of the four runs. We assume in the last round, something went wrong as the first three patches are similar to the ones in the first three rounds.

Run	Patches	Variants	Time
1	6	10, 26, 30, 88, 100, 192	70, 175, 223, 516, 600, 1026
2	6	10, 26, 30, 88, 100, 192	42, 101, 129, 279, 326, 561
3	6	10, 26, 30, 88, 100, 192	67, 165, 211, 466, 545, 942
4	3	10, 26, 30	72, 170, 216

Table 7: Four runs with GP

For the four runs, the first patch for the four runs with GP were found at 70, 42, 67 and 72. Thus, the average time needed to find a patch is 62.75.

jKali

ASTOR was run four times using jKali with the following command

```
java  -cp target/astor-*-jar-with-dependencies.jar fr.inria.main.evolution.AstorMain
-mode jkali -location /home/str/astor/examples/Math-issue-280/
```

In every run of ASTOR using jKali, one patch was found for Variant 9. See Table 8 for the results of the four runs.

Run	Variants	Time
1	9	47
2	9	44
3	9	58
4	9	65

Table 8: Four runs with jKali

For the four runs, the first patch for the four runs with jKali were found at 47, 44, 58 and 65. Thus, the average time needed to find a patch is 53.5.

jMutRepair

ASTOR was run four times using jMutRepair with the following command

```
java  -cp target/astor-*-jar-with-dependencies.jar fr.inria.main.evolution.AstorMain
-mode jMutRepair -location /home/str/astor/examples/Math-issue-280
```

In every run of ASTOR using jMutRepair, one patch was found. See Table 9 for the results of the four runs.

For the four runs, the first patch for the four runs with jMutRepair were found at 724, 683, 1094 and 967. Thus, the average time needed to find a patch is 867.

Run	Variants	Time
1	99	724
2	99	683
3	99	1094
4	99	967

Table 9: Four runs with jMutRepair

ASTOR with jKali is the fastest with an average time of 53.5 to find the first patch. However, it finds only one patch. jMutRepair mode takes the longest time with an average of 867 and also finds only one patch. While none of the modes were able to find the same variant, it can be argued that ASTOR with GP is the most successful, as it is able to find the most patches with an average time of 62.75, which is less than with jMutrepair mode. It is slower than with jKali mode. However, if one wants to find multiple patches, GP mode will perform better.

APPENDIX A

Algorithm 3 Random Fuzzer

```
1: while Time limit not reached do
2:     for Each random input trace do
3:         while Encountered new branch do
4:             sumDistances += compute current branch distance
5:             Count No. of visited unique branches and errors of the input trace
6:             Count Total No. visited branches and errors found
7:         end while
8:         Save branch distance, visited unique branches, and errors of each random input trace
9:     end for
10: end while
11: Log Best input Trace with minimum Branch Distance.
12: Log Total No. visited branches visited and errors found
```

Algorithm 4 Genetic algorithm

```
1: function REPRODUCTION
2:     sorted ← Set with index of individual in all_populations as key and the fitness for each individual as value
3:     Sort sorted on best to worst fitness
4:     for every index in sorted do
5:         best_index ← index
6:         if population.size is smaller than number of BEST_INDIVIDUALS then
7:             Get individual at index
8:             Add individual to best_individuals
9:         end if
10:        Add individual to population
11:        if population.size is equal to POPULATION_SIZE then
12:            break
13:        end if
14:    end for
15:    CROSSOVER
16:    for every best_individual in BEST_INDVIDUAL do
17:        MUTATION(best_individual's Tarantula score)
18:    end for
19: end function
20: function CROSSOVER
21:    i ← 0
22:    while i is less than number of BEST_INDIVIDUALS do
23:        first ← iofpopulation
24:        second ← i + 1ofpopulation
25:        crossover ← randominteger
26:        for index is less than crossover do
27:            Swap index of first and second.
28:        end for
29:        population ← put first on index i
30:        population ← put second on index i + 1
31:    end while
32: end function
33: function MUTATION(tarantula, index)
34:    indiv ← copy individual at index of population
35:    j ← 0
36:    best_mutations ← float array of size OperatorTracker.operators
37:    best_indices ← int array of size OperatorTracker.operators
38:    for entry in tarantula do
39:        if j is smaller than or equal to size of OperatorTracker.operators then
40:            break
41:        end if
42:        best_mutations[j] ← value of entry
43:        best_indices[j] ← key of entry
44:        j ← j + 1
45:    end for
46:    operations_picked ← ROULETTE(best_mutations)
47:    for selected_index in operations_picked do
48:        indiv[best_indices[selected_index]] ← MUTATE(indiv[best_indices[selected_index]])
49:    end for
50:    population ← (index, indiv)
51: end function
```

APPENDIX B

```java
package nl.tudelft.evo;

public class Exam {
    public int grade(boolean cheated, int points) {
        if (points < 0 || points > 100) {
            throw new IllegalArgumentException("invalid points");
        }
        if(cheated) {
            return 1;
        }
        if (points >= 90) {
            return 5;
        } else if (points >= 75) {
            return 4;
        } else if (points >= 60) {
            return 3;
        } else if (points >= 50) {
            return 2;
        } else {
            return 1;
        }
    }
}
```

Figure 4: Original code of project 1

```java
package nl.tudelft.evo;

public class Exam {
    public int grade(boolean cheated, int points) {
        if (points < 0 || points > 100) {
            throw new IllegalArgumentException("invalid points");
        }
        if(cheated) {
            return 1;
        }
        if (points >= 90) {
            return 5;
        } else if (points >= 75) {
            return 4;
        } else if (points >= 60) {
            return 3;
        } else if (points > 50) {
            return 2;
        } else {
            return 1;
        }
    }
}
```

Figure 5: Seed fault that triggered bug in project 1