Exercises

Test Suite Generation

Points	Submission due	Status
6.67 / 25	in 2 days	26.67% (preliminary)

22.22% (preliminary) (5 days ago) GRADED	22.22% (preliminary) (5 days ago) GRADED	24.44% (preliminary) (5 days ago) GRADED	26.67% (preliminary) (5 days ago) GRADED	26.67% (preliminary) (5 days ago) GRADED

Show all results

Tasks:

Test Suite Generation

In the last assignment, we used the multi-objective optimisation algorithm NSGA-II to solve the test suite minimisation problem by trying to achieve two conflicting goals: reducing the number of test cases of a test suite while maintaining high coverage. Yet, we relied on the assumption that the test suite we wanted to optimise already existed in the first place. In practice, software projects spend vast portions of their development time on writing, maintaining and updating test suites. In the Search-Based Software Engineering community, this observation has sparked substantial interest in the research and development of automated test generators. In this assignment, we will seize this idea and implement the Many-Objective Sorting Algorithm (MOSA) to automatically derive test cases that, taken together, form a test suite with high code coverage.

Problem Description

As briefly outlined before, software testing can be a daunting and time-consuming task. In this course, we already implemented techniques that can lower the burden of optimising an existing test suite. Now, we want to take our endeavours one step further and focus our attention on the so-called test case generation problem, which aims to automatically synthesise a test suite for a given class under test (CUT). To solve this problem, we employ MOSA, a slightly modified variant of NSGA-II. Specifically, our goal is to generate a test suite whose test cases maximise branch coverage of the CUT. In other words, every single branch in the CUT is considered an individual optimisation goal. In this problem formulation, the population of the Genetic Algorithm (GA) is made of test cases, and the fitness of an individual is measured in terms of all yet uncovered branches at the same time. As detailed in the literature, the test case generation problem has its peculiarities. Most notably, it poses scalability issues to “traditional” many-objective GAs (such as NSGA-II) due to the so-called dominance resistance phenomenon: the proportion of non-dominated solutions increases exponentially with the number of goals to optimise. As a result, as the number of optimisation goals grows larger and larger, the GA’s search process eventually degenerates into a random one. For instance, it is not uncommon to encounter `Java` classes with hundreds or thousands of branches, each of which is an optimisation goal in its own right. MOSA caters to the specific needs of test case generation by enhancing NSGA-II with a so-called preference criterion. This criterion allows us to assign a preference among non-dominated solutions based on how “close” they come to covering a new, previously uncovered target (in our case, a branch). This way, the number of coverage goals to consider at a time is reduced, and the search is focused only on the still relevant targets. It is important to note that the preference criterion is a stateful operator, i.e., its behaviour depends on the history of the search process, specifically the set of goals which have not been covered yet. This means the preference criterion might rank the same individual differently during different generations. That is, MOSA sacrifices the elitist properties of NSGA-II in exchange for better search focus. Consequently, an optimal solution w.r.t. to a specific goal might simply get lost during the search. To counteract this issue, MOSA maintains a second population, the so-called archive, which is kept separate from the main population and serves as intermediary storage for the best solutions found so far. In detail, after each generation, the archive is updated and supplemented with the best solutions of the main population. In the end, the overall set of best solutions (and therefore our resulting test suite) is formed by the archive.

Chromosome Encoding

In our test suite generation scenario, every chromosome is defined as a test case that consists of a sequence of Java statements that can be executed on the CUT. The sequence of statements can vary in its length but should be limited to 50 statements per test case. Due to the complex nature and feature-richness of the `Java` language, we make the following simplifying assumptions about the CUTs and establish the following

- The first statement of a test case must initialize the CUT using a non-private constructor of the class.
- All following statements are either method calls or field assignments.
- We restrict ourselves to calling dynamic methods of the CUT using this instance and static methods that can always be called independently of the instance of the CUT.
- Statements that assign values to fields only operate on instance fields of the CUT but not on static fields.
- Parameters are directly passed to a method call without storing them in local variables beforehand. For example, foo.bar(1)instead of int i = 1; foo.bar(i).
- When generating input parameters, we limit ourselves to primitive `Java` types, their wrapper classes and Strings. For all other referential data types, it is sufficient to simply pass the `null` reference.
- Numeric input parameters can only range between `-1024` and `1023`.
- For the generation of strings, you can limit yourself to printable ASCII characters between 32 and 126 (inclusive).
- The return value of method calls is always ignored.
- We assume methods do not change the static state of the CUT.
- We assume statements do not throw exceptions. You need not implement exception handling in the generated tests.
- You need not cover inherited methods (e.g., from Object). It is sufficient to target only the methods defined directly in the CUT itself.

Branch Distance

MOSA uses a combination of approach level and normalized branch distance to obtain the fitness value of a test case chromosome. To simplify the assignment, we will consider only the *branch distance*. Let $\mathbb{B} := \{\top, \bot\}$ where $\top$ denotes the `Boolean` value `true`, and $\bot$ the `Boolean` value `false`. Without loss of generality, let

$$\mathbb{P} := \{(x\,\Theta\,y), \neg(x\,\Theta\,y) \mid (x, y) \in \mathbb{Z} \times \mathbb{Z}, \Theta \in \{=, \neq, <, \leq\}\}$$

be the set of possible predicates where $\Theta : \mathbb{Z} \times \mathbb{Z} \to \mathbb{B}$ is a binary relation. For $\beta \in \mathbb{B}$ we define the branch distance $\delta_\beta : \mathbb{P} \to \mathbb{N}_0$ via the following recursive rules:

$$\begin{aligned}
\delta_\top(\rho) &\mapsto 0, & \text{if } \rho \text{ is } \top, (1)\\
\delta_\bot(\rho) &\mapsto 0, & \text{if } \rho \text{ is } \bot, (2)\\
\delta_\bot(x = y) &\mapsto 1, & \text{if } x = y \text{ is } \top, (3)\\
\delta_\bot(x \neq y) &\mapsto |x - y|, & \text{if } x \neq y \text{ is } \top, (4)\\
\delta_\bot(x < y) &\mapsto |x - y|, & \text{if } x < y \text{ is } \top, (5)\\
\delta_\bot(x \leq y) &\mapsto \delta_\bot(x < y) + 1, & \text{if } x \leq y \text{ is } \top, (6)\\
\delta_\top(\neg(x\,\Theta\,y)) &\mapsto \delta_\top(x\,\widehat{\Theta}\,y), & \text{if } \neg(x\,\Theta\,y) \text{ is } \bot, (7)\\
\delta_\bot(\neg(x\,\Theta\,y)) &\mapsto \delta_\bot(x\,\widehat{\Theta}\,y), & \text{if } \neg(x\,\Theta\,y) \text{ is } \top. (8)
\end{aligned}$$

for arbitrary $(x, y) \in \mathbb{Z}$ and $p \in \mathbb{P}$. $\delta_\top(p)$ denotes the distance of p to the `true` branch, and $\delta_\bot(p)$ is the distance of p to the `false` branch. Furthermore, we define

$$(x > y) := (y < x), (x \geq y) := (y \leq x)$$

with $\widehat{\Theta}$ representing the complementary relation of Θ. For example, the complementary relation of $\leq$ is $>$, and the complementary relation of $\neq$ is $=$.

For two references p and q, and referential comparison operators $\{=, \neq\}$, the branch distance can be defined in a similar way, but we change eq. (4) such that

$$\delta_\bot(p \neq q) \mapsto 1, \quad \text{if } p \neq q \text{ is } \top.$$

Then, the remaining rules for reference comparisons are analogous to the ones listed for numbers above.

As an example, let the branch condition be given by $x < y$ where $x = 5$ and $y = 3$. Suppose we want to take the `true` branch, but the branch condition currently evaluates to `false` because 5 is not smaller than 3. As a consequence, control flow currently takes the `false` branch. Therefore, $\delta_\bot(x < y) \mapsto 0$, according to rule (2) of the system. Furthermore, we notice that $\neg(x < y) \iff x \geq y \iff y \leq x$ since the complementary relation of $<$ is $\geq$. According to rules (6) and (8),

$$\begin{aligned}
\delta_\bot(\neg(x < y)) &\mapsto \delta_\bot(x \geq y) = \delta_\bot(y \leq x)\\
&\mapsto \delta_\bot(y < x) + 1\\
&\mapsto |y - x| + 1 = |3 - 5| + 1 = 3
\end{aligned}$$

So far, we have only considered conditional branches. Since we also want to cover branchless methods (i.e., methods without loops, if-then-else statements and so forth) we have to add the “root branches” (i.e., entry points) of methods to the set of coverage targets. Given a method m, the distance δ to its root branch is 0 if

Your Task

You need to implement the two algorithms, *MOSA* and *Random Search*, to solve the test suite generation problem. Similar to assignment two, you are free to choose an appropriate chromosome encoding and corresponding mutation and crossover operators. However, keep in mind that each chromosome should be represented as a sequence of `Java` statements that adhere to the constraints posed above.

Instrumenting Agent

Most of the instrumentation implementation is already complete, with the exception of the `BranchTracer` class. It contains four overloaded `passedBranch()` methods and one overloaded `traceBranchDistance()` method you have to implement. Note that one variant of each method has already been implemented to serve as an example. Each overloaded `passedBranch()` variant handles different comparison operators and shall compute the branch distance as defined above. Once the branch distance calculation is done, `passedBranch()` calls a variant of `traceBranchDistance()` to populate the `distances` hash map of the `BranchTracer` class with the computed branch distance values.

Chromosome Encoding

Once you implemented the branch distance computation, you can now focus on the search algorithms. The first step is to choose a chromosome encoding suitable for the problem at hand. We aim to generate test cases that consist of `Java` statements.

- Implement the `Statement` interface. It represents a statement of a test case in the `Java` programming language. Every statement can execute itself via `Java` Reflection by calling its `run()` method.
- Extend the `Chromosome` abstract class to represent test cases that consist of a sequence of `Statements`. A test case can be executed via its `call()` method, which in turn executes each of the chromosome's statements in sequential order. The result of this execution is a *branch trace*, which tells how close control flow came to reaching the target branches in the CUT. This information can then be processed further for fitness computation.
- Moreover, implement the interface `ChromosomeGenerator` to create random test case chromosomes.

Search Operators

Now that you have defined the problem representation, you can focus on the design of your *search operators*. MOSA uses *mutation* and *crossover* for offspring generation. Implement the two interfaces, `Mutation` and `Crossover`. We place no restrictions on how exactly these operators should behave. This is dependent on the design decisions you made when defining the chromosome representation.

Selection

For the parent selection process, you have to implement the `RankSelection` class. To this, we require you to follow the *rank selection* mechanism as outlined in the lecture slides.

Fitness Function

Our goal is to create a test suite that maximises branch coverage. In this regard, every branch in the CUT is an optimisation goal in its own right. Implement the `FitnessFunction` interface such that, for a given branch, it measures how close control flow came to covering that branch when executing a given test case chromosome based on the *branch distance*. Your implementation can query the measured branch distances by invoking the method `getDistances()` in class `BranchTracer`. It returns a map that maps a branch via its ID to the corresponding branch distance. Subsequently, you will need to create one fitness function per target branch. A branch is represented by the `Branch class in the instrumentation package, and every such branch has a unique integer ID your fitness function could refer to.

Algorithms

Implement Random Search and the MOSA algorithm as presented in the lecture and the literature by implementing the interface `GeneticAlgorithm` for both algorithms.

- For *MOSA*, use *subvector dominance assignment* as a means to estimate population density instead of *crowding distance* assignment. Note that smaller density values imply higher distances to neighbours and are considered better.
- Ensure that both algorithms adhere to the provided `MaxFitnessEvaluations` stopping condition.
- Both algorithms return a list of test case chromosomes from `findSolution()`. In contrast to *NSGA-II* in the previous assignment, *MOSA* does not form this list by taking the chromosomes of the last generation. Instead, it uses its archive to maintain the best solutions found during the search and returns the contents of the archive when the search is finished. You shall do the same for *Random Search*.
- Instantiate your algorithms by implementing the `buildMOSA()` and `buildRandomSearch()` methods in the `AlgorithmBuilder`.

Tips and Remarks

- We provide you with the four test classes `SimpleExample`, `Stack`, `Feature` and `DeepBranches`, which may help you implement and debug the assignment.
- If you find the need to generate random numbers, please do not create your own instances of `java.util.Random`. For the `-s` flag to work properly, use the `Randomness` class instead.
- The `Randomness` class contains the two constants `MAX_INT` and `MIN_INT` that represent the highest possible and lowest possible numeric parameter values, respectively. You can use these constants to generate and modify your test case chromosomes.
- The `branchesToCover` field in `AlgorithmBuilder` stores all branches of the CUT. Thus, every such branch is an optimisation goal of the search, and therefore you need to create one instance of your fitness function for each branch.
- The `testGenerationTarget` field in `AlgorithmBuilder` contains the reflected CUT. The vast majority of coverage goals will be located inside methods which can only be invoked by your tests after an instance of the CUT has been obtained. Use the `testGenerationTarget` field to discover the constructors, methods, and fields the CUT provides, along with the arguments they take. This information is needed by the search algorithm to generate `Statements` and test case chromosomes that operate on the CUT.
- Feel free to create as many (helper) classes and subpackages as you need.
- The tests for *MOSA* and *Random Search* will involve 500 (100 for the `Feature` class) fitness evaluations and 10 experiment repetitions.
- Make sure to add the suffix `Test` to all your unit tests and that you place them in the appropriate `test` directory.
- If you make use of LLM tools such as ChatGPT, upload the prompts you send to the LLM together with the answers you obtained in a folder called `LLM`. Furthermore, ensure to annotate every piece of code you write with the help of LLM tools.
- The execution of all tests will take some time. Thus, we recommend **testing your implementation locally on your machine before pushing it to Artemis**.
  - You do *not* have to push every commit separately to Artemis.
  - If Artemis reports, "No corresponding result available", please wait a couple of minutes (~30 minutes) for the results to appear.
  - In case there are many submissions (e.g. shortly before the submission deadline), it may take longer for the results to appear since your build has to wait in a queue. You can check the current queue length at https://artemis.fim.uni-passau.de/ci-queue-length.html.
- **The Artemis pipeline has a timeout of 15 minutes for each stage (mutation testing, executing your unit tests, executing our functional tests). If the pipeline is interrupted due to the timeout, Artemis will report only partial results for the interrupted stage. ('Test was not executed')**
  - To rate-limit trial-and-error debugging via Artemis, we only run the tests for your latest push. I.e., if you are pushing a second time before you have received the results for the previous push, the tests will only continue to run for the second push and the not-yet-finished first test execution will be aborted.

Functional Tests

1. ✓ **Correct implementation of the branch distance calculation** 6 of 6 tests passing
2. ✓ **Correct implementation of the rank selection operator** 2 of 2 tests passing
3. ✗ **Accepted Random Search results** 0 of 4 tests passing
4. ✗ **Accepted MOSA results** 0 of 4 tests passing

Test Suite Check (20% of points)

1. ✗ **Line Coverage >= 70%** 0 of 1 tests passing
2. ✗ **Branch Coverage >= 60%** 0 of 1 tests passing
3. ✗ **Mutation Score >= 55%** 0 of 1 tests passing

Executing the Application

To guide the search and facilitate coverage measurement, we employ the instrumentation services of the JVM to insert additional instructions into the CUT for branch distance computation. To make this work, the JVM has to be invoked with the flag `-javaagent:<jarpath>=<options>`. Here, `<jarpath>` represents the path to a so-called agent JAR file (i.e., a Java application that instruments the CUT), and `<options>` represent the command line options of the agent. In our case, we use `target/Test-Suite-Generation-jar-with-dependencies.jar` both as `<jarpath>` for the agent, and as main application to execute. As agent `<options>` we use the fully qualified name of the CUT to instrument, such as `de.uni_passau.fim.se2.sbse.suite_generation.examples.SimpleExample`.

Important: the package (here: `de.uni_passau.fim.se2.sbse.suite_generation.examples`) and name (here: `SimpleExample`) of the CUT must match the package and CUT specified via the ---package and ---class command line flags. Putting it all together, the command to invoke the tool with our example CUT and package is as follows:

```
java
-javaagent:target/Test-Suite-Generation-jar-with-
dependencies.jar=de.uni_passau.fim.se2.sbse.suite_generation.examples.SimpleExample
-jar target/Test-Suite-Generation.jar
-c SimpleExample
MOSA
```

For ease of use, the assignment comes with an `.idea` directory that contains a run configuration for executing *MOSA* on the `SimpleExample` class. If you use the `IntelliJ`, the IDE should automatically detect the runtime configuration and offer you the execution of this configuration via the *Run/Debug Configurations* menu.

Exercise details

Release date	Jan 8, 2025 01:00
Submission due	Jan 28, 2025 00:01
Complaint possible	No