|  |
| --- |
| AI for Software Testing and  Reverse Engineering Lab 3 |
| **INTRODUCTION** |
| *Evolutionary algorithms (EAs) are population-based algorithms that are widely used to generate software inputs or tests in an automated fashion.*  *EAs methods use a guided random process for constructing new inputs/tests that try to trigger new code branches. They recombine previously tried inputs/tests guided by a heuristic that measures the proximity to conditions that trigger new branches (approach level and branch distance).*  *EAs, in particular genetic programming, are also used to path existing software. The key idea is that given a set of pre-existing tests, some of which fail, we modify the code until all tests are satisfied. You use the operations from EAs such as crossover and mutation to perform these modifications, and the number of tests satisfied as fitness function.*  *In this lab, you will construct and experiment EAs, understand how they work, and learn how to use them to perform automated software patching.* |
| LEARNING OUTCOMES |
| *After completing this assignment, you will be able to:*     * *Automatically build test sets that maximize branch coverage* * *Use test sets to locate faults in software* * *Build EAs for software patching* * *Use and understand modern EA-based patching tools such as Astor* |
|  |
| INSTRUCTIONS |
| For this assignment, we will again use the RERS problems. Similar to the previous assignment, we have provided you with the instrumentation and you only have to write code for your own solution in *PatchingLab.java*. Before you begin with this assignment, make sure you have pulled the latest changes from the *JavaInstrumenation* repository:  ·   First navigate to the *JavaInstrumentation* folder.  ·   Then pull the latest changes by running the following command: git pull  Instrumenting and compiling a RERS problem is very similar to the previous assignments. The only changes that you need to make to the commands are the following:  ·   Instead of using --type=distance, we now use --type=patch  We then use the following command to instrument a RERS problem:  java -cp target/aistr.jar nl.tudelft.instrumentation.Main --type=patch --file=\*JavaFile\* > \*TargetFile\*    And we use the following command to compile an instrumented RERS problem:  javac -cp target/aistr.jar:. /path/to/instrumented/ProblemX.java    Finally, we use the following command to run an instrumented RERS problem:  java -cp target/aistr.jar:/path/to/folder/containing/instrumented/problem/:. ProblemX Task 1: building a test-set (Optional) To see whether our EA is doing well in the patching, we need  test cases for the RERS problems. As each RERS problem outputs a string based on the given input, we can use the input-output behaviour of the RERS problem to generate test-cases. We have already generated test cases for each RERS problem for you, and provide you with a generator, they are available on Brightspace. However, if you want you can use your own solution from Labs 1-2 to generate the cases yourself (maximizing branch coverage).  You can feel free to play around with the tool that we have used to generate the test cases:   * <https://github.com/TCatshoek/AISTRTestCaseGenerator>   We believe that the test cases that we have generated are complete but we might have overdone it or miss some cases. So if you have a better generator, feel free to generate test cases using your own generator.  The test-cases have the following form:  iA,iB,iC->oXoYoZ  iA,iB,iD->oXoXoZ  …  i.e, the RERS input symbols, separated by “,” characters, followed by “->”, followed by a concatenation of all produced outputs. For this task, you are allowed to build your own test cases for the RERS problem but you have to make sure that you use the same format. *OperatorVisitor.java* contains code to add instrumentation to track the generated outputs:  if (node.getExpression() instanceof MethodCallExpr) {      MethodCallExpr mce = (MethodCallExpr)node.getExpression();      if (node.toString().contains("System.out")) {          this.addCode(node,              new ExpressionStmt(              new MethodCallExpr(              new NameExpr(pathFile),"output",mce.getArguments()              )              ), arg);      }  }  This calls the *output* method in *OperatorTracker.java* whenever something is printed in the standard out. You may use this method but feel free to implement your own method to collect input-output pairs that together cover as many branches as you are able to. Task 2: locating bugs You are provided with buggy versions of each Reachability problem, where only  operators are modified. Your task is to automatically locate the faulty operators using the test sets that were provided to you in Task 1 or use the test sets that you have generated yourself. Of course, one could simply use a code difference method to locate the changes, but in practice you will only have access to failing tests and not a bug-free implementation. Start your task by computing the initial fitness function for the sets you generated on the buggy Reachability problems, i.e., compute the number of failing tests for each problem. The test set is loaded using the *readTests* method.  In *OperatorTracker*, there are two methods that you can use to run an individual test (*runTest*) or all the tests (*runAllTests*) that are listed in the given test set.You can call these methods from *PatchingLab.java*.  First, we compute fault localization to determine which operators are more likely to be faulty by computing how frequently an operator is used in a failing test. *OperatorTracker.java* keeps track of all encountered binary operators in if-statements (>, <, >=, <=, ==, and !=). For every encounter it calls  boolean encounteredOperator(String operator, int left, int right, int operator\_nr)  or  boolean encounteredOperator(String operator, boolean left, boolean right, int operator\_nr)  Where *operator* is the encountered operator, *left* and *right* are the left and right hand sides of the operator and *operator\_nr* is an identifier for the operator (from top to bottom, all operators in the instrumentation are assigned a unique number).  In contrast to labs 1 and 2, in this lab the instrumentation directly influences the functionality of the code. The returned boolean is used instead of the encountered operator in the instrumented if-statement.  Implement the fault localization strategy that uses the tarantula score and use this as part of your genetic algorithm to fix bugs. Describe in pseudocode how you utilise the Tarantula score to patch the buggy operators. Task 3: fixing bugs Use the number of failing tests as a fitness value for a genetic algorithm with fault localization to guide mutations. The instrumented code is set up in such a manner that you do not need to recompile the code in order to test mutations of operators (which only modified these operators) without the need to recompile. The current operator list is maintained in OperatorTracker:  OperatorTracker.operators[operator\_nr]  The result of each test is stored in *OperatorTracker.test\_result.* Use this to compute a fitness function. Your algorithm needs to contain the following components:   * Mutation is a random change or one operator into another, guided by fault localization. * Implement any selection strategy, such as tournament selection. * Use as search heuristic the number of passing tests.   Experiment with mutation rates and show the performance of your genetic algorithm by plotting a convergence graph (fitness vs. time) and what percentage of faults you have correctly patched for Reachability problems that you have chosen to run your algorithm on. Task 4: Genetic Programming using ASTOR You are asked to use the tool ASTOR (<https://github.com/SpoonLabs/astor>) to find a patch for buggy versions of the RERS problems. ASTOR is already available in the docker container we provided for this course. Detailed instructions on how to run ASTOR on the three RERS problems above are also posted on Brightspace.  **Step 1**: First, you have to generate test cases for the original version (without bugs) of the three RERS problems above. You can use the test cases we provide. These are included in the packages on Brightspace, generated using EvoSuite.  **Step 2**: Your job is to run ASTOR to find a patch to the target bug. For the assignment, you need to run ASTOR using Genetic Programming as patch engine. You will set this engine using the parameter -**mode jgenprog**. More details information on how to rung ASTOR are available on Brightspace.  **Step3:** For each buggy version of the RERS problem, run ASTOR once. Then, report on statistics regarding:   * The time it needed to find the patches (if any). * How many patches were found by ASTOR.   Then, manually analyze the generated patches and answer the following questions:   1. Does ASTOR generate a meaningful patch? Use a diff between the original RERS problems, their buggy versions, and the patched version. 2. Did ASTOR generated meaningful patches? Is genetic programming effective at finding patches, why (not)? |
| **RESOURCES** |
| Slides from Lectures 5  Study:   1. Le Goues, Claire, et al. "Genprog: A generic method for automatic software repair." Ieee transactions on software engineering 38.1 (2012): 54-72 2. Martinez, Matias, and Martin Monperrus. "Astor: A program repair library for java." Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, 2016. |
| **PRODUCTS** |
| A small report of max 2 A4 pages answering the questions from tasks 2-3-4, visualizations are allowed in at most 2 extra A4 pages.  An archive (tar/zip) containing the code for computing the results.  Make sure you have also provided some instructions on how to run your code. |
| **ASSESSMENT CRITERIA** |
| You can either pass or fail this assignment. We expect everyone to pass. You have to complete all lab assignments. Submissions of which the report text does not fit into 2 A4s will not be evaluated.  Code that does not compile/run will not be evaluated.  Your work will be evaluated based on its completeness (having done all tasks), the correctness of the implementation, and demonstration that you understand the results in the analysis. When deemed insufficient, you will receive feedback and will be given a one-week grace period to fix any shortcomings. |
| **SUPERVISION AND HELP** |
| There will be lab sessions  every Wednesday, where the teachers and TAs will be available  to answer any questions you may have. The preferred way to ask questions is through  Mattermost. |
| **SUBMISSION AND FEEDBACK** |
| The submission is through Brightspace. You will receive feedback within one week after the deadline. |