Amendment:

- Fixing the typo in the sample output in question 2. It should be “(D, B)” instead of (B, D) in the line “bug: D in scope2, pair: (B, D), support: 4, confidence: 80.00%”.
- Add clarification to question 2:
    - Only need to consider pairs (i.e., itemset of size 2)
    - The thresholds support and confidence can be increased as you see fit

Automatic test case generation is one of the widely used techniques that generate a large number of test cases to detect the differences between existing and required behaviour of a given software system. This question explores the usefulness and readability of automated generated test cases. We use code coverage and the number of bugs detected to evaluate the quality of a test suite (i.e., a set of test cases). Project `binarytree` is a very simple implementation of Binary Tree with Java. Using this project as a starting point, finish the following tasks. The project is maintained by maven (https://maven.apache.org/index.html), which manages a project's build, reporting and documentation from a central piece of information. We use `jacoco` (have already been configured in maven pom.xml for you) to measure the code coverage.

In the q1/, you have three different folders, i.e., binarytree, binarytree-randoop, and binarytree-evosuite, which will be used for questions (a), (b), and (c), respectively.

You may want to work on a Linux/Ubuntu/MacOs machine or use Windows Subsystem for Linux (WSL) on Windows for this question. Note: for Evosuite, you would need to install and use JDK 8 or earlier.

Useful Materials:

- Randoop manual: https://randoop.github.io/randoop/manual/index.html
- Evosuite manual: http://www.evosuite.org/documentation/maven-plugin/
- Maven: https://maven.apache.org/guides/getting-started/maven-in-five-minutes.html

(a) Open `binarytree` folder, and write as many test cases as you can in Junit test class `src/test/java/york/eecs/bt/BinaryTreeTest.java`. Please make sure you have at least 10 test cases and cover all the functions in class `src/main/java/york/eecs/bt/BinaryTree.java`.

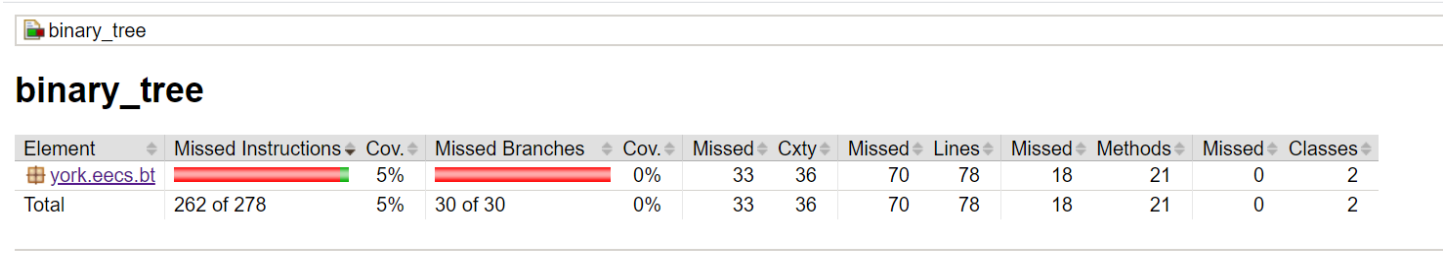
- Step 1: compile the project:
  ```
  cd binarytree
  mvn clean
  mvn compile
  ```

- Step 2: write your own test cases:
  At least 10 tests which cover all functions in BinaryTree class (Remember to add comments to indicate what each test case is testing, you will not get maximum points without explaining your test cases)

- Step 3: run your new tests:
  ```
  mvn test
  ```

- Step 4: check code coverage of your test cases:
  `cd target/site/jacoco` and open `index.html`, you will find the coverage information page.

For example, the following two pages show the coverage information of original test cases.

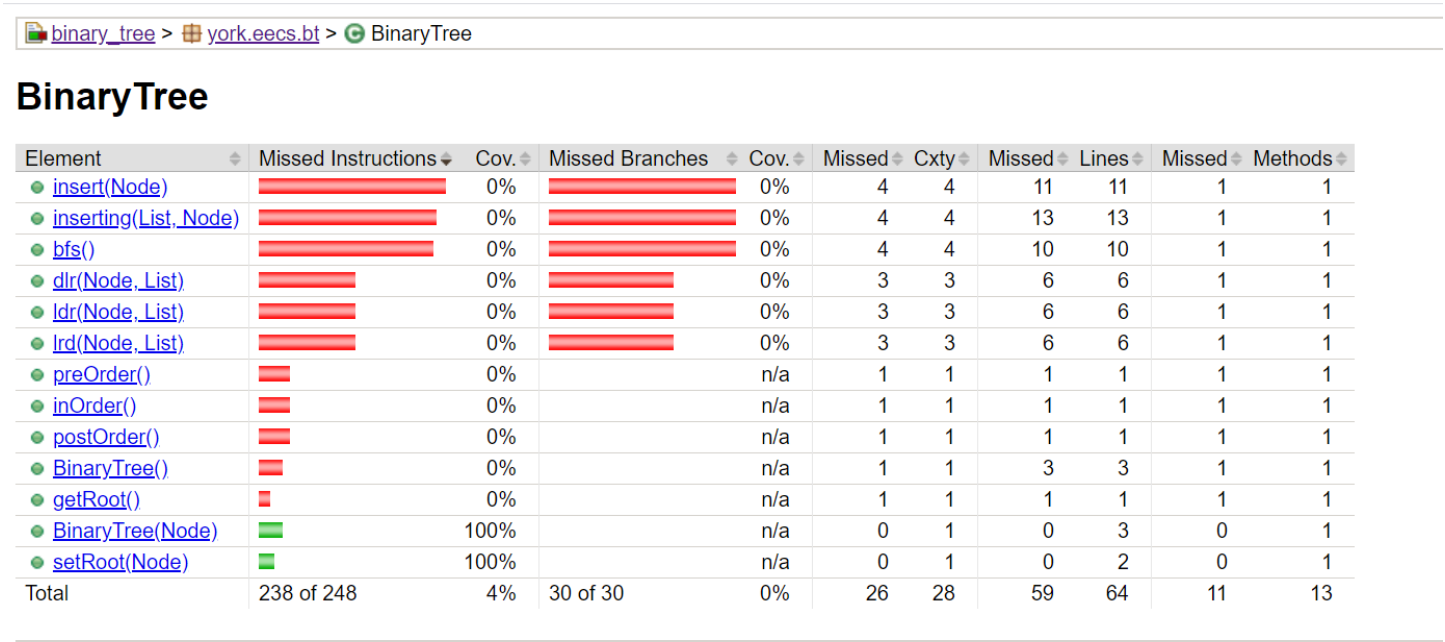


binary_tree

binary_tree

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
york.eecs.bt		5%		0%	33	36	70	78	18	21	0	2
Total	262 of 278	5%	30 of 30	0%	33	36	70	78	18	21	0	2

You can further unfold BinaryTree class.



binary_tree > york.eecs.bt > BinaryTree

BinaryTree

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
insert(Node)		0%		0%	4	4	11	11	1	1
inserting(List, Node)		0%		0%	4	4	13	13	1	1
bfs()		0%		0%	4	4	10	10	1	1
dlr(Node, List)		0%		0%	3	3	6	6	1	1
ldr(Node, List)		0%		0%	3	3	6	6	1	1
lrd(Node, List)		0%		0%	3	3	6	6	1	1
preOrder()		0%		n/a	1	1	1	1	1	1
inOrder()		0%		n/a	1	1	1	1	1	1
postOrder()		0%		n/a	1	1	1	1	1	1
BinaryTree()		0%		n/a	1	1	3	3	1	1
getRoot()		0%		n/a	1	1	1	1	1	1
BinaryTree(Node)		100%		n/a	0	1	0	3	0	1
setRoot(Node)		100%		n/a	0	1	0	2	0	1
Total	238 of 248	4%	30 of 30	0%	26	28	59	64	11	13

(b) Open `binarytree-randoop` folder, and generate test cases with `randoop`, please use the following commands to run `randoop` and generate test cases.

- Step 1: compile the project:
  ```
  cd binarytree-randoop
  mvn clean
  mvn compile
  ```
- Step 2: generate test cases for class BinaryTree:
  ```
  java -classpath ./target/classes/:./randoop-lib/randoop-all-4.2.1.jar
  randoop.main.Main gentests --testclass=york.eecs.bt.BinaryTree --output-limit=500
  ```
- Step 3: clean temporary internal files:
  ```
  rm *.class
  ```
- Step 4: edit the generated test files, i.e., add the package name (``package york.eecs.bt;`") to the

head of `RegressionTest0.java` and `ErrorTest0.java` (if these two files are generated).

- Step 5: move these two generated test files into your test folder, i.e., `src/test/java/york/eecs/bt/`.
- Step 6: run the tests generated by `randoop`:
  `mvn test -Dtest=york.eecs.bt.RegressionTest0` or
  `mvn test -Dtest=york.eecs.bt.ErrorTest0`
- Step 7: check code coverage:
  `cd target/site/jacoco` and open `index.html`

(c) Open `binarytree-evosuite` folder, and generate test cases with `evosuite`, please use the following commands to run `evosuite` and generate test cases.

- Step 1: compile the project:
  ```
  cd binarytree-evosuite
  mvn clean
  mvn compile
  ```
- Step 2: generate test cases by using `evosuite` (we have already configured `evosuite` in maven):
  `mvn evosuite:generate`
- Step 3: export your test cases into your project (i.e., move to `src/test/java/york/eecs/bt/`):
  `mvn evosuite:export`
- Step 4: edit the generated test files, i.e., ``*_ESTest.java" files as followed. Do not modify the ``*_scaffolding.java" files.

  ```
  @RunWith(EvoRunner.class) @EvoRunnerParameters(mockJVMNonDeterminism =
  true, useVFS = true, useVNET = true, resetStaticState = true,
  separateClassLoader = true, useJEE = true)
  ```

  In the above configuration, change `separateClassLoader = true` into `separateClassLoader = false`.
- Step 5: run the tests generated by `evosuite`:
  `mvn test -Dtest=york.eecs.bt.BinaryTree_ESTest or`
  `mvn test -Dtest=york.eecs.bt.Node_ESTest`
- Step 6: check code coverage:
  `cd target/site/jacoco` and open `index.html`

(d) Compare the different test suites written by yourself, generated by `randoop`, and generated by `evosuite`. Describe what you find regarding the coverage, readability, effectiveness, usefulness, etc.

(e) Write a Python program to automate randoop-based test case generation described in (b), i.e., from Step 1 to Step 6.

Your submission should include:

1. Test cases and their coverage information (e.g., figures) from question (a).
2. Test cases generated by `randoop` and their coverage information (e.g., figures) from question (b).
3. Test cases generated by `evosuite` and their coverage information (e.g., figures) from question (c).
4. Your analysis of these different test suites.
5. Your Python/Bash script for question (e).

Once you are done with coding and running test case generation, zip your working folder q1 as follows:

```
zip -r firstname_lastname_q1.zip q1
```

Build Rule-Based Bug Detectors

(a) Inferring Likely Invariants for Bug Detection

Take a look at the following contrived code segment.

```
void scope1() {
  A(); B(); C(); D();
}
void scope2() {
  A(); C(); D();
}
void scope3() {
  A(); B(); B();
}
void scope4() {
  B(); D(); scope1();
}
void scope5() {
  B(); D(); A();
}
void scope6() {
  B(); D();
}
```

We can learn that function `A` and function `B` are called together 3 times in function `scope1`, `scope3`, and `scope5`. Function `A` is called 4 times in functions `scope1`, `scope2`, `scope3`, and `scope5`. In this case, since function `A` and `B` are called together 3 times, we infer that the one time that function `A` is called without `B` in `scope2` is likely a bug (an unlikely occurrence could represent an abnormality). Note that we only count `B` once in `scope3` even though `scope3` calls `B` 2 times.

We define `support({A,B})` as the number of times a pair `{A,B}` of functions appear together. Therefore, `support({A,B})` is 3. We define `confidence({A,B},{A})` as `support({A,B})/support({A})`, which is 3/4.

We set the threshold for support and confidence to be `T_SUPPORT` and `T_CONFIDENCE`, whose default values are three and 65% (can be configurable). To avoid false positives, we only print bugs with confidence `T_CONFIDENCE` or more and with pair support `T_SUPPORT` times or more. For example, function `B` is called 5 times in function `scope1`, `scope3`, `scope4`, `scope5`, and `scope6`. The 2 times that function `B` is called alone are not printed as a bug as the confidence is only 60% (`support(A,B)/support(B)` = 3/5), lower than the `T_THRESHOLD`, which is 65%. Please note that `support(A,B)` and `support(B,A)` are both equal 3.

For question (a), we do not expand `scope1` in `scope4` to the four functions `A`, `B`, `C`, and `D`. Match function names only. Do not consider function parameters. For example, `scope1()` and `scope1(int)` are considered the same function. Note that you only need to consider pairs of methods. So only need to look for itemset of max size of 2.

The sample output with the default support and confidence thresholds should be:

```
bug: A in scope2, pair: (A, B), support: 3, confidence: 75.00%
bug: A in scope3, pair: (A, D), support: 3, confidence: 75.00%
bug: B in scope3, pair: (B, D), support: 4, confidence: 80.00%
bug: D in scope2, pair: (D, B), support: 4, confidence: 80.00%
```

Generate call graphs using BCEL framework. The Byte Code Engineering Library (Apache Commons BCEL) is intended to give users a convenient way to analyze, create, and manipulate (binary) Java class files (those ending with .class). Classes are represented by objects which contain all the symbolic information of the given class: methods, fields and byte code instructions, in particular. Such objects can be read from an existing file, be transformed by a program (e.g. a class loader at run-time) and written to a file again. An even more interesting application is the creation of classes from scratch at run-time. The Byte Code Engineering Library (BCEL) may be also useful if you want to learn about the Java Virtual Machine (JVM) and the format of Java .class files.

The details of BCEL are here: https://commons.apache.org/proper/commons-bcel/manual/bcel-api.html

Given a bytecode file, we can use BCEL to generate call graphs and analyze the textual call graphs to generate function pairs and detect bugs. The *java-callgraph* (github.com/gousiosg/java-callgraph) tool (an application of BCEL) is given to you for generating call graphs from bytecode. You can find the tool in /q2.

Instruction of run \java-callgraph" tool:

You can use the following command to generate the call graph for a given jar file in the text format.

```
java -jar javacg-0.1-SNAPSHOT -static.jar {path -to-a-jar-file}
```

e.g., to generate call graph for project “\commons-math3-3.6.1.jar":

```
java -jar javacg-0.1-SNAPSHOT -static.jar proj/commons-math3-3.6.1.jar
```

Given the following Java source file:

```
public class Test_Null {
      public static void main(String [] args){
            String str = null;
            System.out.println(str.toLowerCase());
      }
}
```

A call graph generated from the bytecode of the above Java code in the text format is shown as follows:

```
C:Test_Null Test_Null
C:Test_Null java.lang.Object
C:Test_Null java.lang.System
C:Test_Null java.lang.String
C:Test_Null java.io.PrintStream
M:Test_Null:<init>() (O)java.lang.Object:<init>()
M:Test_Null:main(java.lang.String[]) (M)java.lang.String:toLowerCase()
M:Test_Null:main(java.lang.String[]) (M)java.io.PrintStream:println(java.lang.String)
```

The following explanation should help you understand the call graph:

- **Format for classes (starting with \C"):**

  `C:class1 class2`

  This means that some method(s) in class1 called some method(s) in class2.

- **Format for methods (starting with \M"):**

  `M:class1:<method1>(arg types) (typeofcall)class2:<method2>(arg types)`

  The line means that method1 of class1 called method2 of class2.

- A valid method call sequence is: `[java.lang.String:toLowerCase, java.io.PrintStream:println]`, please ignore the `java.lang.Object:<init>` and `main()` methods as these methods are essential initialization methods or entry points of Java applications. So in short, you only need to worry about the virtual method calls (M). Noted that methods with the same name but with different parameters will be counted as a single method.

Performance. We will evaluate the performance and scalability of your detector on a pass/fail basis. The largest program that we test will contain up to 50k nodes and 100k edges. Each test run will be given a timeout of 10 minutes. A timed-out test run will receive zero points. We do not recommend using multi-threading because it only provides a linear gain in performance, doing so will not earn you maximum points.

Hints:

- **Important!** Do not use string comparisons because it will greatly impact the performance. Use a hashing method to store and retrieve your mappings.
- Focus on method call pairs to calculate your support.
- Minimize the number of nested loops. Pruning an iteration in the outer loop is more beneficial than in the inner loop.

Your tasks:

- Implement your code for inferring support of function pairs and detecting bugs with these supports in the given python file: detector.py (in folder /q2).
- Your detector.py takes three parameters: `-jar <path to jar> -sup <support> -c <confidence>`, an example is as follows:

  ```
  detector.py -jar proj/opennlp-tools-1.9.2.jar -sup 5 -c 0.75
  ```

- The output will be the potential bugs detected by the detector with the following format:

  ```
  bug: X in M, pair: (X, Y), support: S, confidence: C
  ```

  Examples are:

  ```
  bug: B in method2, pair: (A, B), support: 3, confidence: 75.00%
  bug: D in method3, pair: (B, D), support: 4, confidence: 80.00%
  ```

(b) Finding and Explaining False Positives (20 points)

Table 1: Details of the experimental subjects.

Project	Version	Description	#method	#class
commons-math3	3.6.1	a library of lightweight, self-contained mathematics and statistics components	36K	11K
solr	8.4.1	an open-source enterprise-search platform, written in Java, from the Apache	49K	21K
weka	3.6.1	open source machine learning software	164K	29K
deeplearning4j	1.0.0	a deep learning programming library written for Java	0.8K	0.3K
opennlp	1.9.2	a machine learning based toolkit for the processing of natural language text	29K	11K
mahout	0.9	a distributed linear algebra framework	25K	13K

You will apply your code to the latest versions of real-world projects listed in table 1 (you can find their jar files in folder /proj). Examine the output of your code for (a) (i.e., detector.py) on each of the six projects. The values of `T_SUPPORT` and `T_CONFIDENCE` are 3 and 65%, respectively. You can increase these thresholds as you see fit.

Do you think all the reported bugs are real bugs? There are some false positives. A false positive is where a line says “\bug ..." in your output, but there is nothing wrong with the corresponding code. Write down two different fundamental reasons why false positives occur in general. What effect is changing `T_SUPPORT` and

`T_CONFIDENCE` have on the false positives rate?

Identify 2 pairs of functions that are false positives for each project. Explain why you think each of the two pairs that you selected are false positives. The two pairs should have a combined total of at least 5 \bug ..." line locations (e.g., the first pair can have 4 and the second pair can have 6). For example, the following sample output contains 4 locations regarding 3 pairs: `(A, B) (A, D)` and `(B, D)`. You do not have to provide explanations for all 5 locations, but you will want to use at least one location to discuss each unique pair. Briefly explain why the pair is a false positive regarding the source code.

```
bug: A in scope2, pair: (A, B), support: 3, confidence: 75.00%
bug: A in scope3, pair: (A, D), support: 3, confidence: 75.00%
bug: B in scope3, pair: (B, D), support: 4, confidence: 80.00%
bug: D in scope2, pair: (B, D), support: 4, confidence: 80.00%
```

The source code of each experiment projects has been provided under the folder proj/source/. You can find more detailed context information for each reported bug. This will also help you explain the reason for false positives. Remember to report the time it takes your tool to finish the detection run for each project (note that you have to keep the single-threaded performance under 10 minutes. A reduced mark will be given if you have to resort to multithreaded).

Using Static Bug Detection Tool: Pylint. For this question, you will be using the static code analysis tool Pylint (https://www.pylint.org/) to find possible defects in your own code, i.e., detector.py from question 2.

Run Pylint on your own code from Q2 (a). See instructions on https://www.pylint.org/ to analyze your code. You learn more from having the tool find defects in code that you wrote, fixing the problem and seeing the defect go away. Discuss two of the most interesting bugs detected by Pylint using up to 1 page (talk about why Pylint detected the bugs, whether Pylint was correct, how you fixed the bug, and the lesson learned). If Pylint finds no bugs, what are the possible reasons?

Once you are done with using Pylint on your detector.py zip up the working folder as follows:

```
zip -r firstname_lastname_q3.zip q3
```