EECS4313 Software Engineering Testing
Final Project (100 points), Version 1
Test Case Prioritization For Regression Testing

Instructors: Song Wang
Release Date: April 18, 2020

Due: 11:59 PM, May 2nd, 2020

In this project, you are expected to build test case prioritization models for scheduling test execution during the regression testing scenarios. You are required to apply static code analysis, e.g., collecting coverage between test cases and source code, generating the diff between two versions, etc., to build effective test case prioritization models with the dataset provided.

Policies

- You are required to work on your own for this assignment. No group partners are allowed.
- When you submit your work, you claim that it is solely the work of you. Therefore, it is considered as an violation of academic integrity if you copy or share any parts of your code during any stages of your development.
- Your (submitted or un-submitted) solution to this assignment (which is not revealed to the public) remains the property of the EECS department. Do not distribute or share your code in any public media (e.g., a non-private Github repository) in any way, shape, or form. The department reserves the right to take necessary actions upon found violations of this policy.
- Emailing your solutions to the instruction or TAs will not be acceptable.

Submitting Your Work

- create a folder named "EECS4313_Exam".
- put all the reports/result analysis from each part into one PDF file, named **EECS4313_exam_report.pdf**.
- create subfolders with names "**part_I**", "**part_II**", and "**part_III**".
- create a subfolder name "**data**", put your experiment data in this folder.
- do not submit your temporary intermediate data, which could be huge, make sure the Instructor/TAs can run your code to replicate these results.
- make sure the Instructor/TAs can run your code based on the above file structure without any manual modification.

```
zip -r EECS4313_Exam.zip EECS4313_Exam
submit 4313 exam EECS4313_Exam.zip
```

Table 1: The Evaluated project. **LVersion** is the last version of a project. **CVersion** is the current version under test. **Lang** is the programming language used for the project. **#File** is the number of source classes. **#TClass** is the number of test classes in each project. **#TMethod** is the number of test methods in each project. **#Fault** is the number of bugs in the current version.

Project	LVersion	CVersion	Lang	#File	#TClass	#TMethod	#Fault
mahout	2.11	2.12	Java	13K	143	2.5K	22

Background

A typical Test Case Prioritization (TCP) technique reorders the execution sequence of test cases based on a certain objective, e.g., the coverage of test cases. Specifically, TCP can be formally defined as follows: given a test suite T and the set of its all possible permutations PT, TCP techniques aim to find a permutation $P^{'} \in PT$ that $(\forall P^{''})\ (P^{''} \in PT)\ (P^{''} \neq P^{'}),\ f(P^{'}) \geq\ f(P^{''})$, where f is the objective function.

CovTCP: Most TCPs leverage coverage information, e.g., dynamic code coverage (i.e., the source code covered when test cases were executed) from the last run of test cases or static code coverage from static code analysis on current version under test. The commonly used coverage criteria include statement, method, and branch coverages. In this project, we choose to examine static code coverage on statement and method levels.

SimTCP: Another important TCP category is based on the similarity, which first calculates the similarity between each test case and the changed source code between current version and last version, and then rank test cases for prioritization based on the similarity values. Note that, for these test cases that have the same similarity, **SimTCP** often randomly orders them.

For **CovTCP** techniques, there are two widely used prioritization strategies, i.e., total strategy and additional strategy. The total coverage strategy schedules the execution order of test cases based on the total number of statements or methods covered by these test cases. Whereas, the additional coverage strategy reorders the execution sequence of test cases based on the number of statements or methods that are not covered by already ordered test cases but covered by the unordered test cases.

Evaluation Metrics

We use the Average Percentage Fault Detected (APFD), a widely used metric for evaluating the performance of TCP techniques. APFD measures the average percentage of faults detected over the life of a test suite, and is defined by the following formula:

$$APFD = 1 - \frac{\sum_{i=1}^{num_f} TF_i}{num_t \times num_f} + \frac{1}{2 \times num_t} \tag{1}$$

where, num_t denotes the total number of test cases, num_f denotes the total number of detected faults, and TF_i $(1 \leq i \leq num_f)$ denotes the smallest number of test cases in sequence that need to be run in order to expose the fault i. APFD values range from 0 to 1. For any given test suite, its num_t and num_f are fixed. The higher APFD value signals that the average value of TF_i is lower and thus represents a higher fault-detection rate.

In this project, you are expected to compare the performance of different TCPs by using APFD.

Experiment Data

In this project, we use one project from your assignment A2 as the experiment data, which is shown in Table 1. We use two consecutive versions, one is the last version (i.e., **LVersion**) that has executed all the test cases in the last run, another is current version (i.e., **CVersion**) that under test.

For the current version in each project, there are multiple faults. **In this project, we define a fault as a failed test case.** We expect you to build different types of TCP algorithms and prioritize the test cases to find these faults early (i.e., with a higher APFD).

Definitions

- **Test Class:** A class with the keyword "Test" in its class name will be considered as a Test Class, for example:

  mallet.extract.tests.TestPerDocumentF1Evaluator

- **Test Case:** A test case is a method in a test class that containing the keyword "test" in its method name. Usually in a test class, there exists many test cases. For example:

  mallet.extract.tests.TestPerDocumentF1Evaluator:testPerFieldEval()

- **Faults/Bugs:** A fault is a failed test case.

(I)- Collect Static Coverage (20 points)

Your first task is to collect static code coverage of each test case in a project under test, which will be used to build the coverage based TCPs.

Figure 1 shows an example class (i.e., `Account`) and the test class (i.e., `AccountTest`) contains six test cases. By using static analysis, we can build the call graph starting from test cases. Figure 2 shows the call graph of each test case. Test case `t1` directly calls two methods, i.e., `transfer()` and `withdraw()`. Method `transfer()` directly calls two methods, i.e., `sendto()` and `getBalance()`, and `withdraw()` directly calls methods `sendBalance()` and `getBalance()`. By doing the inter-procedural analysis, we can finally get the method-level code coverage for each test case, which is shown in Table 2.

In this project, we reuse the call graph generation library we used in your A2, i.e., `javacg -0.1-SNAPSHOT -static.jar` to collect the method-level coverage information.

```
public class Account extends BasicAccount{
 public Account(String acnt, double amt) {
   super(acnt, amt);}
 public void sendto(String account) {
  // send money to target account}
 public boolean deposit(double amt) {
  if (amt > 0) {
   setBalance(getBalance() + amt);
   return true;}
  else
   return false;}
 public boolean withdraw(double amt) {
  if (getBalance() >= amt && amt > 0) {
    setBalance(getBalance() - amt);
    return true; }
  else
   return false;}
 public boolean transfer(double amt, String anotherAcnt) {
   double fee = amt * 0.01;
  if (getBalance() >= amt + fee && amt > 0) {
    withdraw(amt + fee);
    sendto(anotherAcnt);
    return true; }
  else
   return false; }
}
```

```
public void test1(){
 a.transfer(50.0, "user2
 a.withdraw(40.0);
 //Assertions}
public void test2(){
 a.withdraw(50.0);
 a.sendto( "user2 ");
 //Assertions}
public void test3(){
 a.withdraw(10.0);
 a.getBalance();
 //Assertions}
public void test4(){
 a.deposit(30.0);
 //Assertions}
public void test5(){
 a.getBalance();
 //Assertions}
public void test6(){
 a.setBalance(30.0);
 //Assertions}
```

Figure 1: Example class *Account* and the related test cases.

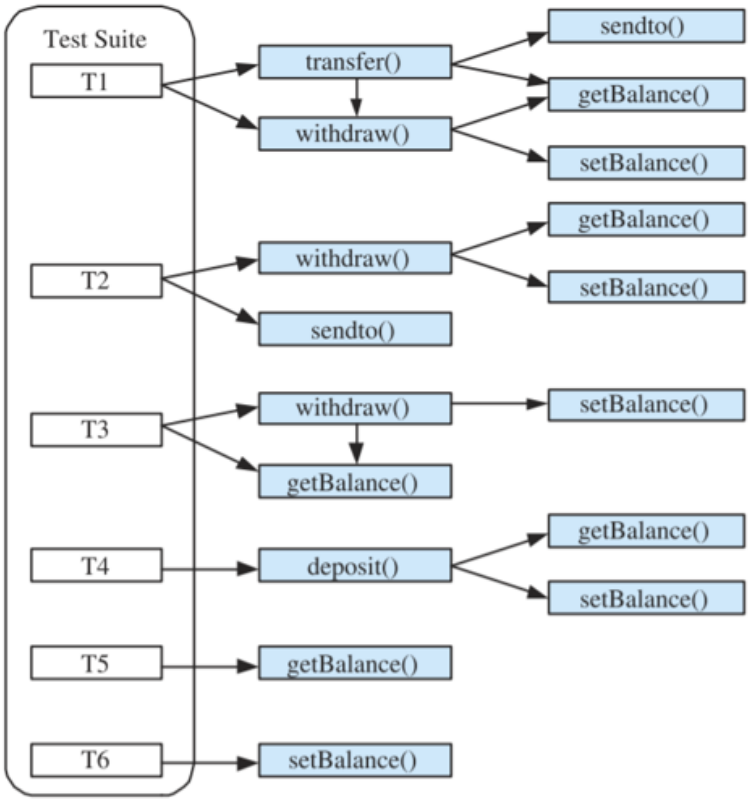

Figure 2: Method-level coverage of each test case.

Your task:

- Follow the example showed above to build you own method-level static code coverage in **coverage.py**. Your tool should support one parameter, i.e., project path. For example:

  coverage.py -path mahout.jar

- The method-level coverage **only contains methods declared in this project (hint: use the package name style of a project to filter out invalid methods)**, all other methods from third-party libraries should be discarded.

- The output of your tool will be the test cases (with alphabetical order) and the inferred method-level coverage with the following format:

  TestClass.test_case_name: **[Class.method1, ...]**

  AccountTest.test1: [Account.transfer(double, String), Account.withdraw(double), Account.sendto(String), Account.getBalance(), Account.setBalance(double)]

- You can put the output into a text file with the following format:

  mahout-coverage.txt

Performance. We will evaluate the performance and scalability of your program on a pass/fail basis. The largest program that we test will contain up to 50k methods. Each test case will be given a timeout of 5 minutes. A timed-out test case will receive zero points.

Table 2: Method-level coverage of each test case.

Test Case	Method-level coverage
test1	Account.transfer(double, String), Account.withdraw(double), Account.sendto(String), Account.getBalance(), Account.setBalance(double)
test2	Account.withdraw(double), Account.sendto(String), Account.getBalance(), Account.setBalance(double)
test3	Account.withdraw(double), Account.getBalance(), Account.setBalance(double)
test4	Account.deposit(double), Account.getBalance(), Account.setBalance(double)
test5	Account.getBalance()
test6	Account.setBalance(double)

(II)- Build Static Coverage based TCP (40 points)

With the method coverage for each test case you collect in Part I, your next task is to build the coverage based TCP algorithms with different strategy, i.e., total and additional strategies.

Your task:

- Build you own static code coverage based TCP algorithms with `total strategy` in **tcp_total.py**. Your tool should support one parameter, i.e., coverage path. For example:

  tcp_total.py -path mahout-coverage.txt

- Build you own static code coverage based TCP algorithms with `additional strategy` in **tcp_additional.py**. Your tool should support one parameter, i.e., coverage path. For example:

  tcp_additional.py -path mahout-coverage.txt

- The output of your tools on a project contains 1) a list of ordered test cases and 2) the `APFD` value. You can put the ordered test cases and APFD in files, e.g.,

  mahout-total-result.txt,

  mahout-total-apfd.txt,

  mahout-additional-result.txt, and

  mahout-additional-apfd.txt

In your report of this part, please examine your above TCP algorithms on the project and then report the performance with figures or tables.

(III)- Build Similarity based TCP (40 points)

In this part, you are expected to build a similarity based TCP algorithm. Specific instead of using coverage, we use the similarity between test cases (**representing by the coverage information you collected in Part I**) and the changes of source code files (**excluding test classes**) between two versions. We have already generated the patch in **/data/mahout.patch** for you.

Specifically, the **mahout.patch** file contains all the changes of source code files. You are expected to use a list of tokenized string tokens to represent the patch and all the test cases. And then use cosine-similarity[1] to calculate the similarity between each test case and the **mahout.patch**.

In the patch file, we only focus on the diff information of java files, i.e., with the suffix ".java". The changes share the following format, e.g., starting from "diff –git ...", you can use this hint to break the patch file into multiple changes and filter out changes that are not on java files.

[1] `https://en.wikipedia.org/wiki/Cosine_similarity`

```
diff --git a/examples/src/main/java/org/apache/mahout/classifier/NewsgroupHelper.java b/examples/src/
    main/java/org/apache/mahout/classifier/NewsgroupHelper.java
index 5cec51c0b..3674a5779 100644
--- a/examples/src/main/java/org/apache/mahout/classifier/NewsgroupHelper.java
+++ b/examples/src/main/java/org/apache/mahout/classifier/NewsgroupHelper.java
```

For tokenizing the **patch** file (after filtering out non-java-related changes), you can treat it as pure text file and collect all the tokens in the patch. For test cases, we use the coverage information collected in Part I to represent their content.

You can customize your tokenization choice. Please described the details in your report. For example, suppose we a method call in the **patch** file or test cases, i.e., Account.transfer(double, String), we can simply tokenize them into [Account, transfer, double, String].

Your task:

- Build you similarity based TCP algorithms in **tcp_similarity.py**. Your tool should support two parameters, i.e., **-cov**: coverage information, **-pl:** path to the patch. For example:

  tcp_similarity.py -cov mahout-coverage.txt -pl mahout-patch

- The output of your tools on a project contains 1) a list of ordered test cases and 2) the `APFD` value. You can put the ordered test cases and APFD in files, e.g.,

  mahout-sim-result.txt and

  mahout-sim-apfd.txt.

In your report of this part, please compare the performance of your coverage based TCP algorithms to the similarity based TCPs. You can use figures and tables to discuss your results. Reasonable explanations are expected to help understand the results of your comparison.