Course Project Final Report

CS4612 - Program Analysis Fall '17

Tarantula Program Fault Localization

Ali Ahsan - 18100071
Muhammad Zain Qasmi - 18100276

Supervisor: Sir Junaid Haroon Siddiqui

Department of Computer Science
Lahore University of Management Sciences (LUMS)

December 28th, 2017

Abstract

Despite of so much resources being spent on developing and testing programs, programs today with their ever increasing scale and complexity tend to contain bugs which are difficult to track and fix - making it impractical for programmers to manually locate them. Hence, in this project we will focus on ways to make fault localization easy for programmers.

Background

Locating the bugs or errors in a program is one of the most frustrating tasks for a programmer. We were motivated by the problems a programmer has to face in identifying the bugs and in getting rid of them. It's still a very tiring task despite the amount of resources spent on automated testing and the availability of tools such as debuggers in IDEs.

Traditional methods have relied on analyzing memory dumps which requires processing a large amount of data or on inserting print statements by making intelligent to catch the bug. This demands a good understanding of the program and it's not feasible for large-scale programs.
We were interested in techniques which can rank program statements in terms of their suspiciousness. This led us to the tarantula automatic fault localization technique.

Description

The idea is to build on existing machine learning and data mining based fault localization techniques by increasing their scope and effectiveness to make debugging more feasible for programmers.

Traditional methods of fault localization relied either on analyzing memory dumps which inarguably requires processing a huge cache of data or on inserting print statements by making intelligent guesses to center on the bug. However, this requires the user to have a good understanding of the program and is not feasible for large-scale programs even with the use of tools such as Microsoft Visual Studio debugger. Since the primary problem of isolating code based on its susceptibility to fault is still unresolved.

Most modern fault localization methods work by focusing on prioritizing suspicious code based on its likelihood of containing faults which the programmers manually checks on later whether it indeed contains bugs - since reducing this search space is critical to easily isolating bugs.

The critical part of this will be the mechanism/algorithm used to give a “likelihood” score to every statement in the code. One approach by Tarantula was to calculate the ratio of test cases passed and test cases failed by a statement and sort them by this score. However, this technique is limited by the number of test cases available - multiple failed test cases let the technique leverage the richer information base. Also, this likelihood approach ignores the program structure and dependencies between program entities rather focusing only on

tracing execution for failed cases.

There were few other options available as well such as Nearest Neighbour, Cause Transitions and Set Union and Set Intersection. Set Union, Set Intersection and Nearest Neighbour were rejected since the paper on fault localization techniques[1] argued they are less effective because of their sensitivity to particular test suites. Cause transition was an intriguing choice since it performs a binary search between a passing test case and failing test case. It does perform better than the three mentioned before but empirical studies have shown that Tarantula outperforms Cause Transition consistently. [3]

Project Specification

We implemented the famous fault localization technique known as Tarantula such that an automated Python program can be used to identify bugs in a given python script. In our implementation we computed the suspiciousness score of a line of code's likelihood for bug using the formula given in the paper i.e.

$$suspiciousness(e) = \frac{\frac{failed(e)}{total\ failed}}{\frac{passed(e)}{total\ passed} + \frac{failed(e)}{total\ failed}}$$

A dictionary datastructure was used to cater the ranks of each of line of code using "key" as line number and its "value" is the ranking based on suspiciousness score. The suspiciousness score for each line was calculated and sorted such that the line with highest score was given a rank of 1 and should be evaluated first and foremost for bug checking. If that statement is found to not be faulty, then the statement with the next rank is evaluated.

To test which lines were hit by the code, we used Python's settrace() method from sys library which takes three arguments - the stack frame; event; and an argument frame which is the current stack frame. An event simply returns a 'call' or 'line' string etcetera based on which the function decides what to do such as in case of a 'call' string the trace function is invoked and returns a reference to the local trace function to be used in that scope. So whenever any line is hit, it is reported by the tracing function which we could catch in our dictionary to store key value pairs of statements hit against the particular test case that was being run. We preferred settrace method over the python library coverage.py since we could directly use the coverage information given by settrace method inside our script, whereas for coverage it would have been an indirect method to first run on the command line and then feed the results to our script.

A basic arithmetic function then used the values stored in this dictionary datastructure to evaluate the suspiciousness score of each line while also taking into account whether that particular test failed failed or passed i.e. by measuring the number of passed test cases hit and the number of failed test cases hit by each line of code then pushing these as arguments to the suspiciousness function to calculate each line of code's score. The score may vary from 0 to 1 (1 being the most suspicious).

We wrote unit tests against three simple python programs against which our program was evaluated. Python's unittest library was used to test the programs and generate test data. For example, we wrote a mid() function that takes three numbers as input and should return the median of the three values as output. Our program successfully determined line 7 as the most suspicious statement where the bug was expected to be found and present. We also used Python's hypothesis library to compute test cases which fail the program. This was particularly helpful because we were able to derive the test cases which would not meet the assertion and hence saved our time from manually figuring out the cases. We're attaching the generateFailMid.py file which generates a failing test case for mid function.

The ranking algorithm simply takes the suspiciousness values of all the lines of code and sorts them in descending order based on their suspiciousness score. Also, much effort was spent on automating our tool such that any python program with arbitrary number of arguments and arbitrary number of test cases can be processed by it.

Future Work

We have mostly tested our tool on small python scripts spanning at most one file. We are not sure how our tool would perform on programs spanning multiple files and over thousands of lines of code.

Furthermore, test cases can be auto-generated against the given programs instead of being supplied by the user.

Lastly, for users convenience a browser based interactive GUI could be developed that loads the test program and test cases into the client and displays them on the screen.

Source Code

- **Test Case 1**

```
def mid(x,y,z):
      m = z
      if (y<z):
            if(x<y):
                  m = y
            elif (x<z):
                  m = y       #bug
      else:
            if(x>y):
                  m = y
            elif (x>z):
                  m = x
```

```
        print "Middle Number is: " + str(m)
```

Our program successfully identifies the bug at line 7 with a suspiciousness rating of 0.833

- **Test Case 2**

```
def is_prime(number):
    if number < 0:
        return True         #bug
    if number in (0, 1):
        return False
    for element in range(2, number):
        if number % element == 0:
            return False
    return True
```

Our program successfully identifies the bug at line 3 with a suspiciousness rating of 1.0

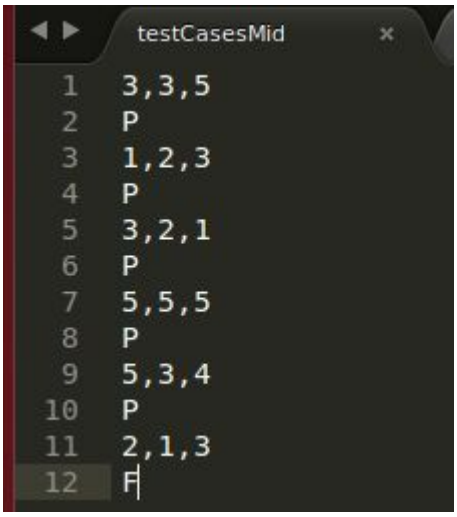
- **Test Case 3**

```
def maximum(x,y,z):
     m = 0       #bug
     if (x > m):
          m = x
     if (y > x):
          m = y
     if (z > y):
          m = z
     #print "Middle Number is: " + str(m)
     return m
```

Our program gave a false positive about the bug being at line 6 with a suspiciousness rating of 0.833. The actual bug was at line 2 and our program assigned it a suspiciousness rating of 0.5.

How to Run

- Copy your python test program(s) and its test case(s) argument(s) in the project's root directory.
- Test cases arguments have to be in csv format with 'P' for pass and 'F' for fail for every consecutive case as shown below:



- Run the program from terminal using the following command:

```
python tarantula.p <first arg> <second arg>
```

First Arg: Your python test program file name
Second Arg: Your test cases file name

Example:

```
python tarantula.py is_prime.py testCasesPrime
```

- You should expect a similar output as shown below:

```
zainqasmi@zainqasmi:~/Desktop/Tarantula2$ python tarantula_v3.py is_prime.py testCasesPrime
Top 10 most suspicious lines
Line    Suspiciousness  Rank    Line of Code
3       1.0             1               return True             #bug
1       0.5             3       def is_prime(number):
2       0.5             3           if number < 0:
4       0.0             9           if number in (0, 1):
5       0.0             9               return False
6       0.0             9           for element in range(2, number):
7       0.0             9               if number % element == 0:
8       0.0             9                   return False
9       0.0             9           return True
Detailed report exported to output.txt
zainqasmi@zainqasmi:~/Desktop/Tarantula2$
```

- You may find the detailed report in file output.txt at the project's root directory

Bibliography

“A Survey of Software Fault Localization” - November, 2009
http://www.utdallas.edu/~ewong/fault-localization-survey.pdf

“Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique” - 2005
http://spideruci.org/papers/jones05.pdf [3]

“DeLLIS” - July, 2009
https://www.researchgate.net/publication/221391301_DeLLIS_A_data_mining_process_for_fault_localization

“On the Accuracy of Spectrum-based Fault Localization”
http://ieeexplore.ieee.org/document/4344104/#full-text-section

“Visualization of Test Information to Assist Fault Localization” - May, 2002
https://www.cc.gatech.edu/~stasko/papers/icse02.pdf

“Coverage.py”
https://coverage.readthedocs.io/en/coverage-4.4.2/

“Unit Testing Framework”
https://docs.python.org/2/library/unittest.html

“Hypothesis testing Library”
https://github.com/HypothesisWorks/hypothesis-python