Mutation Testing of Source Code

Software Testing CS-731

IMT2020049 Vaibhav Thapliyal
IMT2020099 Karanveer Singh

Abstract:

A Python code for Analyzing Sunburst Charts is used to carryout mutation testing using mutpy library.

Introduction:

Mutation testing is a software testing technique that assesses the quality of a test suite by introducing small, intentional defects (mutations) into the source code and checking if the test suite can detect these mutations. The primary goal of mutation testing is to identify areas of weakness in the test suite, helping developers improve their tests and, consequently, the overall reliability of the software.

Mutation Operators:

Introduction of Mutants: Mutation testing involves creating variations or mutants in the original source code by applying specific mutation operators. These operators simulate common programming errors, such as changing arithmetic operations, negating conditions, or inserting incorrect values.

Mutant Types: Mutation operators fall into different categories, each representing a type of potential error. For example, arithmetic mutants, logical mutants, and relational mutants target specific types of operations.

Mutant Generation:

Creating Mutants: The mutation testing tool generates mutants by applying mutation operators to the source code. Each mutant represents a slightly modified version of the original code.

Mutant Set: The set of mutants comprises the original code plus all variations created by applying mutation operators.

Test Execution:

Running the Test Suite: The existing test suite is executed on both the original code and the mutants. This involves running all test cases that are part of the suite.

Observing Test Results: The testing tool monitors the results of each test case for both the original code and the mutants. The goal is to determine whether the test suite can distinguish between correct behavior and the introduced mutations.

Mutation Analysis:

Classifying Mutants: Mutants are classified based on whether they are killed or survived. A killed mutant is one for which the test suite detects a fault, while a surviving mutant indicates that the test suite did not identify the introduced defect.

Mutation Score: The mutation score is calculated as the percentage of killed mutants out of the total mutants. A higher mutation score indicates a more effective test suite.

Reports and Feedback:

Generating Reports: Mutation testing tools provide reports detailing the results of the mutation analysis. These reports often include information on the mutation score, a list of killed mutants, and any surviving mutants.

Feedback for Improvement: Developers use the information from the reports to improve the test suite. They can add new test cases, modify existing ones, or address areas where the test suite failed to detect mutations.

Mutation testing provides a thorough evaluation of the effectiveness of a test suite by considering not only whether tests pass but also whether they can detect subtle errors introduced through mutations. While it is a powerful technique for improving test quality, it can be computationally expensive due to the large number of mutants generated, especially in complex codebases. As a result, mutation testing is often used in conjunction with other testing methods to strike a balance between effectiveness and computational resources.

Source Code:

[Github link]

Sunburst-Chart-Analyzer

SunburstChartAnalyzer is a tool for extracting hierarchical data from images of sunburst charts and representing it as a tree data structure.

Description
Sunburst charts are a popular way to visualize hierarchical data using concentric circles and annular sectors. Manually extracting the data from sunburst chart images can be tedious.

This project aims to automate the data extraction process using computer vision and image processing techniques. Given an image, it first detects whether it contains a sunburst chart. If so, it extracts the text from the chart image and detects the hierarchical levels encoded in the chart geometry. Finally, it constructs a tree data structure representing the hierarchy and data labels from the sunburst chart.

The workflow consists of:

Chart classification - Use machine learning models like SVM and CNN to detect if the image contains a sunburst chart Component extraction

Circle detection to find chart center

Text detection using OCR to extract labels

Text removal while retaining background

Line detection using probabilistic Hough transform to identify hierarchy levels

Hierarchical data extraction - Construct tree structure by assigning parent-child relationships based on geometry

Unit Testing:

Tools used :

Tools used to test the code were Mutmut tool for python

Tests run :

Tests were run on a file containing 8 classes which are unit tested called Misc_function.py
The test file containing the test cases was test1.py

Testing results :

Using mutmut tool we ran unit tests on Misc_function.py file, which contained 8 functions.
The mutant operators applied were
AOR - arithmetic operator replacement
ASR - assignment operator replacement
COI - conditional operator insertion
ROR - relational operator replacement
LCR - logical connector replacement

Out of 111 mutants, 94 were killed and 17 survived giving a mutation score of 84.7 percent.

The coverage was 60% with 506 nodes covered out of 855 total nodes.

```
[0.00100 s] killed by test_ang_av3 (test1.UnitTest)
[*] Mutation score [12.92347 s]: 84.7%
   - all: 111
   - killed: 94 (84.7%)
   - survived: 17 (15.3%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
[*] Coverage: 506 of 855 AST nodes (59.2%)
```

```
coverage:
  all_nodes: 855
  covered_nodes: 506
mutation_score: 84.68468468468468
```

We can find the unit test report file in the github link provided.

Individual mutant tests like AOR, COI and ROR give mutation score as -

AOR : 77.8%
COI : 100%
ROR : 86.5%

Integration Testing:

Integration testing carried out on three functions which call functions from the Misc_function.py file. Namely:

- distr_between_midpoints(x1, y1, x2, y2, x3, y3, x4, y4)
- if_angle_inside_ret_avg(x,y,z)
- in_circle(x, y, r,x_circle,y_circle)

For each of the functions, 5 test cases were created, 15 in total.

Integration Parameter Exchange

Each parameter in the tested method is replaced with another parameter of the same type. The experiments are attached as below

```
def distr_between_midpoints(x1, y1, x2, y2, x3, y3, x4, y4):
    mid_1 = Misc_function.midpoint(x1, y1, x2, y2)
#   mid_2 = Misc_function.midpoint(x3, y3, x4, y4)
    mid_2 = Misc_function.midpoint(y3, x3, x4, y4)

    return Misc_function.distance_formula(mid_1[0], mid_1[1], mid_2[0], mid_2[1])
```

```
----------------------------------------------------------------------
Ran 5 tests in 0.002s

FAILED (failures=5)
```

All the test cases failed to pass hence the mutant was killed

```
# def midpoint(x1, y1, x2, y2)
def midpoint(x1, x2, y1, y2):
    x_mid = int((x1 + x2)/2)
    y_mid = int((y1 + y2)/2)
    return (x_mid, y_mid)
```

Changing variable in midpoint() function which is called by distr_between_midpoints() function in a different file.

```
----------------------------------------------------------------------
Ran 5 tests in 0.003s

FAILED (failures=5)
```

All the test cases failed to pass hence the mutant was killed.

Integration Method call deletion

Here each method call is deleted, and if the method returns a value which is used in an expression, it is replaced with an appropriate constant value.

```
def if_angle_inside_ret_avg(x,y,z):
#    if Misc_function.angle_inside(x,y,z):
    if False:
        return Misc_function.angle_avg(x,y)
```

Here, the function call is deleted and replaced with constant bool value.

```
----------------------------------------------------------------------
Ran 5 tests in 0.003s

FAILED (failures=5)
```

The test cases failed to pass hence the mutant was killed

```
def if_angle_inside_ret_avg(x,y,z):
    if Misc_function.angle_inside(x,y,z):
    #    return Misc_function.angle_avg(x,y)
        return 55.0
```

Here the function call angle_avg() was replaced by a constant value.

```
----------------------------------------------------------------------
Ran 5 tests in 0.002s

FAILED (failures=4)
```

4 tests failed and and one passed, giving us a mutation score of 80%

Integration Return Expression Modification

Each expression in each return method is modified by applying UOI or AOR operators.

```
def del_redundant_lines(line_levels):
    for i in line_levels:
        rem_ind = []
        for j in line_levels[i]:
            for k in line_levels[i]:
                if(k != j):
                    if(abs(k[1]-j[1]) < 3.5):
                        if(distance_formula(k[0][0][0], k[0][0][1], k[0][1][0], k[0][1][1]) < distance_formula(j[0][0][0], j[0][0][1
                            rem_ind.append(k)
                        elif(distance_formula(k[0][0][0], k[0][0][1], k[0][1][0], k[0][1][1]) == distance_formula(j[0][0][0], j[0][0
                            line_levels[i].remove(k)
        for m in rem_ind:
            if(m in line_levels[i]):
                line_levels[i].remove(m)

    return line_levels
```

Test case were run for del_redundant_lines() function where the return value from distance_formula() function were mutated.

```
def distance_formula(x1, y1, x2, y2):
    dist = math.sqrt(pow(x1-x2, 2)+pow(y1-y2, 2))
#   return dist
    return -dist
```

```
----------------------------------------------------------------------
Ran 3 tests in 0.061s

FAILED (failures=3)
```

Here all three test cases failed to pass hence the mutant was killed.

```
def distance_formula(x1, y1, x2, y2):
    dist = math.sqrt(pow(x1-x2, 2)+pow(y1-y2, 2))
#   return dist
    return dist + 340
```

Here the return expression is modified using AOR

```
----------------------------------------------------------------------
Ran 3 tests in 0.003s

OK
```

In this case we can see the mutant survived, this is because since del_redundant_line() uses distance_formula to compare quantities, adding or subtracting doesnt affect the results. This is an **equivalent mutant** for this case.

Tests were run for the given function in the next experiment

```python
def distr_between_midpoints(x1, y1, x2, y2, x3, y3, x4, y4):
    mid_1 = Misc_function.midpoint(x1, y1, x2, y2)
    mid_2 = Misc_function.midpoint(x3, y3, x4, y4)

    return Misc_function.distance_formula(mid_1[0], mid_1[1], mid_2[0], mid_2[1])
```

And midpoint() function was mutated.

```python
def midpoint(x1, y1, x2, y2):
    x_mid = int((x1 + x2)/2)
    y_mid = int((y1 + y2)/2)
    # return (x_mid, y_mid)
    return (x_mid + 5, y_mid*3)
```

Here the return expression is modified using AOR

```
----------------------------------------------------------------------
Ran 5 tests in 0.003s

FAILED (failures=5)
```

All test cases failed to pass hence the mutant was killed in this case.