An Experimental Comparison of Unit Test Criteria

Hadis Ghafouri

Introduction

Unit testing is increasingly becoming important for many software projects.

The overall quality of the software has a strong impact on the economic success of software systems and the companies that sell software.

The increasing use of agile processes has an especially strong impact on testing.

Programmers are expected to do more, and better, unit testing.

This paper presents experimental data to help compare cost benefit tradeoffs among four test criteria.

Introduction

Test criterion

- a rule or collection of rules that, given a software artifact, imposes test requirements that must be met by tests.
- Test criteria make it more likely that testers will find faults in the program.
- provides greater assurance that the software is of high quality and reliability.
- can be based on lots of software artifacts, including formal specifications, requirements.
- most unit testing is based on the program source.

Test requirements

- specific elements of software artifacts that must be satisfied or covered.
- ex: test requirement for statement coverage is “Execute statement 7.”

The criteria describes the test requirements for the artifact in a complete and unambiguous manner.

Introduction

Many test criteria are based on graphs.

This research compares four test criteria: edge-pair, prime path, all-uses data flow and mutation.

The criteria are compared on:

1. The number of tests that were needed to satisfy the criteria.
2. The ability of those tests to find faults.

The findings in this paper provide evidence that can help practical testers choose which test criteria to use and when.

Criteria Comparisons

Our ability to compare test criteria theoretically is limited.

Two relationships are commonly used to compare test criteria.

1.PROBBETTER relationship

> A test criterion C1 is PROBBETTER than C2 for a program *P* if a randomly selected test set T that satisfies C1 is more "likely" to detect a failure than a randomly selected test set that satisfies C2.

Example :let's say we have two different test criteria for testing a calculator application:

C1: All mathematical operations (+, -, *, /) are tested with input values ranging from -1000 to 1000.

C2: Only addition and subtraction operations are tested with input values ranging from -10 to 10.

Let's say we run 100 random tests for each criterion, and find that 10 failures were detected with C1 and only 3 failures were detected with C2.

we can conclude that C1 is PROBBETTER than C2, because it is more likely to detect failures than C2.

2.PROBSUBSUMES relationship

> A test criterion C1 PROBSUBSUMES C2 for a program *P* if a test set *T* satisfies the requirements of C1, then it is likely to satisfy the requirements of C2 as well.

If C1 PROBSUBSUMES C2, C1 is said to be more "difficult" to satisfy than C2.

Criteria Comparisons

These relationships were used to compare mutation and all-uses in several papers.

Paper 1

- compared mutation and all-uses using the PROBBETTER relationship.
- They found that mutation was more effective for five of their nine subjects.
- all-uses was more effective for two.
- there was no difference for the other two.

Paper 2.

- compared mutation and all-uses with both PROBBETTER and PROBSUBSUMES.
- They found that mutation-adequate test sets were closer to satisfying the all-uses criterion and detected more faults.

This paper uses the PROBBETTER relationship and adds two criteria that have not been compared with mutation and all-uses:

- edge-pair.
- prime path coverage.

Test Criteria Used

Directed graphs form the foundation for many test criteria.

graph G is

- a set N of nodes, where N ≠ ∅.
- a set N0 of initial nodes, where N0 ⊆ N and N0 ≠ ∅.
- a set Nf of final nodes, where Nf ⊆ N and Nf ≠ ∅.
- a set E of edges, where E is a subset of N×N.

Test requirements are *satisfied* by

1. *visiting* specific nodes or edges.
2. *touring* specific paths or subpaths.

Test path

- represents the execution of a test case on a graph.
- Test paths must start at an initial node and end at a final node.
- A test path p tours a subpath q if q is a subpath of p.

Test Criteria Used

Edge-pair (EP) test criterion

- Defined for finite state machines.
- Tests must tour each reachable sub-path of length less than or equal to 2 in *G*.
- The qualification "less than or equal to 2" is included specifically to ensure that edge-pair coverage subsumes edge coverage and node coverage in graphs that have only one edge or only one node.

Prime path (PP) test criterion

- is to ensure strong coverage of loops without requiring an infinite number of paths.
- Tests must tour each prime path in the graph G.
- It subsumes node coverage, edge coverage and edge-pair coverage.

Simple path

- A path from node ni to nj is simple if no node appears more than once in the path.
- exception: the first and last nodes may be identical.
- simple paths have no internal loops, although the entire path itself may be a loop.

Test Criteria Used

Prime paths

- maximal length simple paths, from node ni to nj if it does not appear as a proper sub-path of any other simple path.
- Prime paths do not have any internal loops, although the entire path may be a loop.

Data flow criteria

- require tests that tour subpaths from specific definitions of variables to specific uses.
- Nodes where a variable is assigned a value are called *definitions* (or *defs*).
- Nodes where the value of a variable is accessed are called *uses*.
- A definition d for a variable x *reaches* a use u if there is a path from *d* to *u* that has no other definitions of x (*def-clear*).

All-uses (AU) criteria

- requires tests to tour at least one sub-path from each definition to each reachable use.

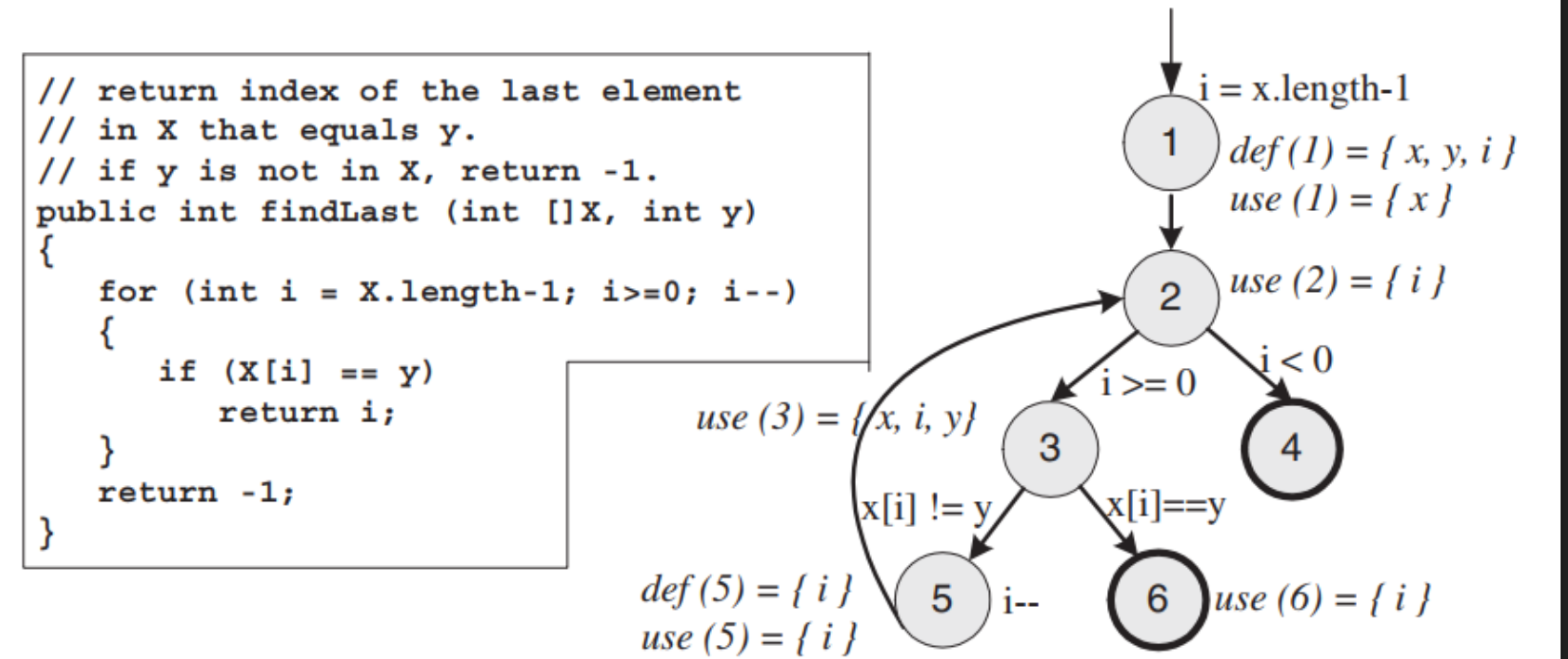
findLast(): control flow graph, test requirements

Edge-Pairs = { [1, 2, 3], [1, 2, 4], [2, 3, 5], [2, 3, 6], [3, 5, 2], [5, 2, 3], [5, 2, 4] }

Prime Paths = { [1, 2, 4], [1, 2, 3, 6], [1, 2, 3, 5], [2, 3, 5, 2], [3, 5, 2, 3], [3, 5, 2, 4], [5, 2, 3, 6], [5, 2, 3, 5] }

Def Use Pairs = { (x, 1, 1), (x, 1, 3), (y, 1, 3), (i, 1, 2), (i, 1, 3), (i, 1, 5), (i, 1, 6), (i, 5, 2), (i, 5, 3), (i, 5, 6), (i, 5, 5) }

```
// return index of the last element
// in X that equals y.
// if y is not in X, return -1.
public int findLast (int []X, int y)
{
   for (int i = X.length-1; i>=0; i--)
   {
      if (X[i] == y)
         return i;
   }
   return -1;
}
```



Test Criteria Used: Mutation coverage

Mutation coverage

Mutation Testing(MT) is also known as fault-based testing, error-based testing.

mutation testing is a software testing type that is based on changes or mutations.

- small changes are introduced into the source code to check whether the defined test cases can detect errors in the code.
- The ideal case is that none of the test cases should pass.
- If the test passes, then it means that there is an error in the code. We say that the mutant (the modified version of our code) lived.
- If the test fails, then there is no error in the code, and the mutant was killed.
- Our goal is to kill all mutants.
- The more mutants we can kill, the higher the quality of our tests.

Mutation score

- This is a score based on the number of mutants.

$$\text{Mutation score} = \frac{\text{Number of killed mutants}}{\text{Total number of mutants (survived and killed)}} * 100\%$$

Mutation examples

value mutation
changing the parameter and/or constant values, usually by +/- 1.

Original code

```
let arr = [2,3,4,5]
for(let i=0; i<arr.length; i++){
    if(i%2===0){
        console.log(i*2)
    }
}
```

Mutant code

```
let arr = [2,3,4,5]
for(let i=1; i<arr.length; i++){
    if(i%2===0){
        console.log(i*2)
    }
}
```

Statement mutation
1. delete or duplicate a statement in a code block.
2. rearrange statements in a code block.

Original code

```
let arr = [2,3,4,5]
for(let i=0; i<arr.length; i++){
    if(i%2===0){
        console.log(i*2)
    }
}
```

Mutant code

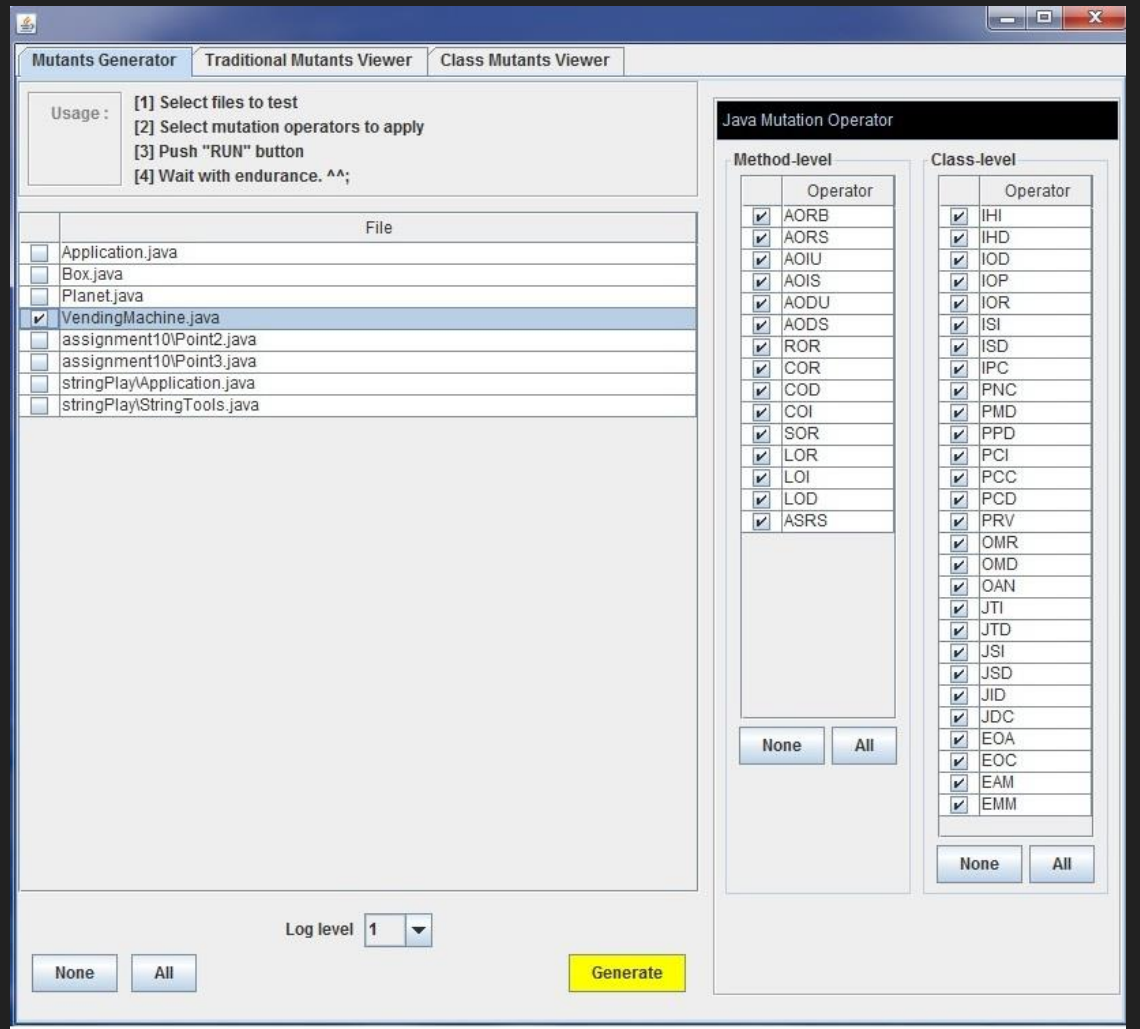
```
let arr = [2,3,4,5]
for(let i=0; i<arr.length; i++){
    if(i%2===0){
        console.log(i*2)
        console.log(i*2)
    }
}
```

MuJava

We generated test values by hand and used muJava to evaluate the tests.

MuJava

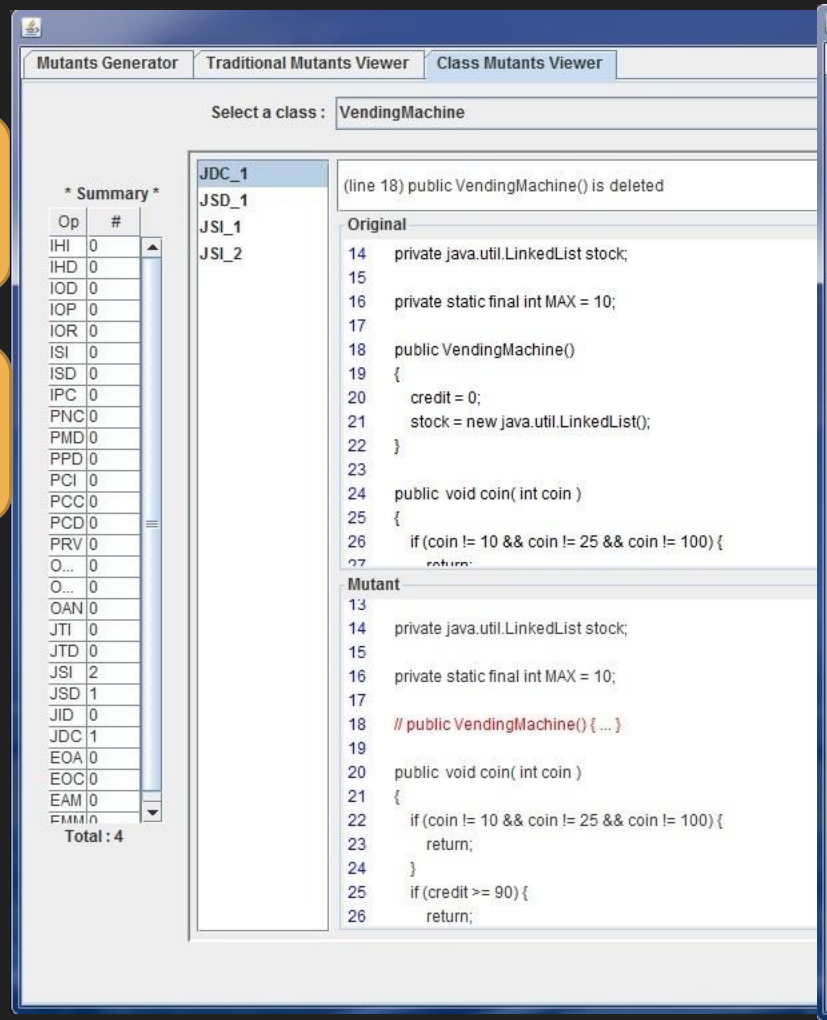
- is a software testing tool.
- is used for generating and detecting faults in Java programs.
- It works by mutating the original code of a Java program to create a set of mutant versions, each containing a single fault.
- These faults are then used to test the robustness of the program, as well as to assess the effectiveness of test cases.
- includes tools for visualizing the results of the testing process.
- It generates mutants, runs tester supplied tests against the mutants, and computes the mutation score.

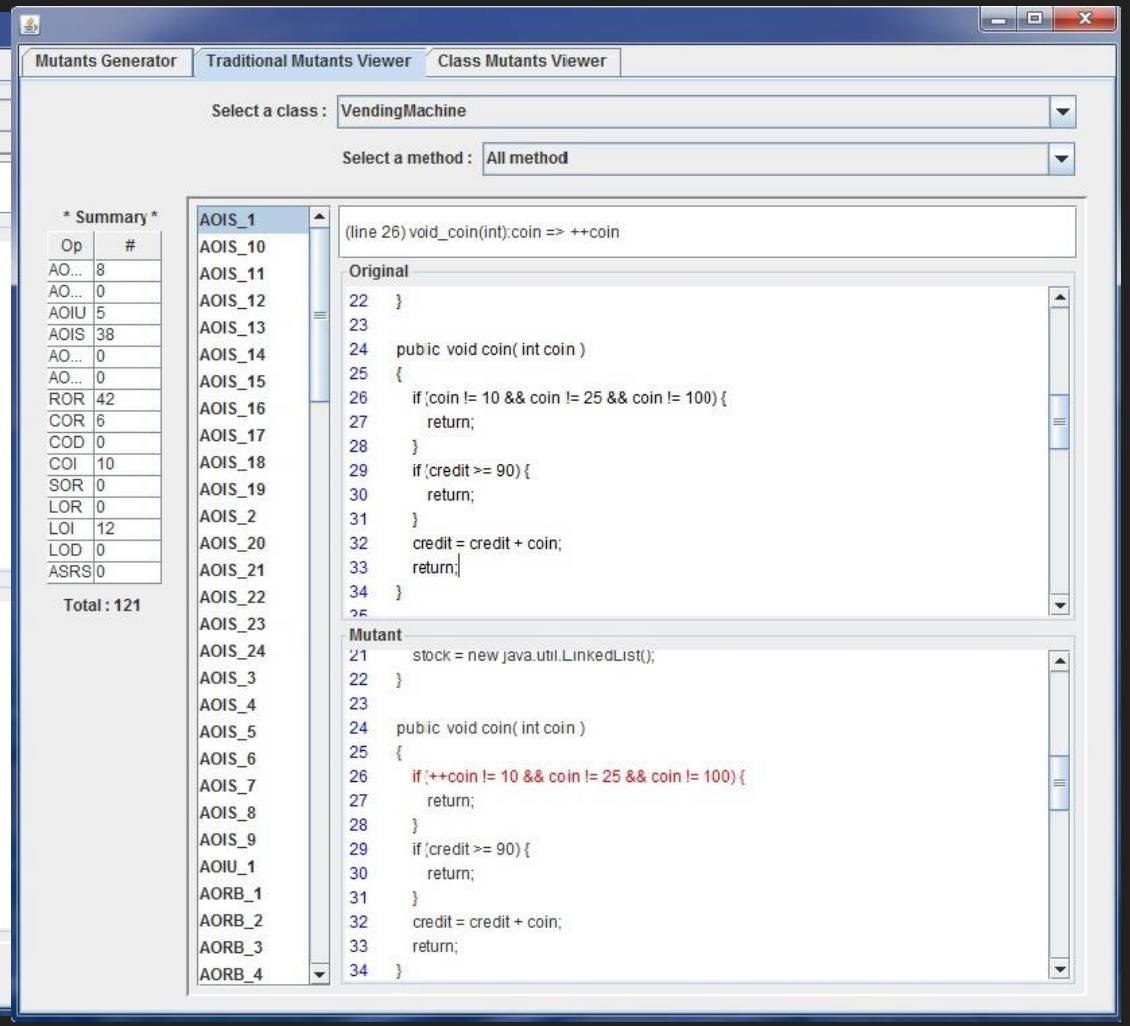


MuJava

The mutated code is then compiled and executed using a set of test cases.

the results are compared to those obtained from the original code.

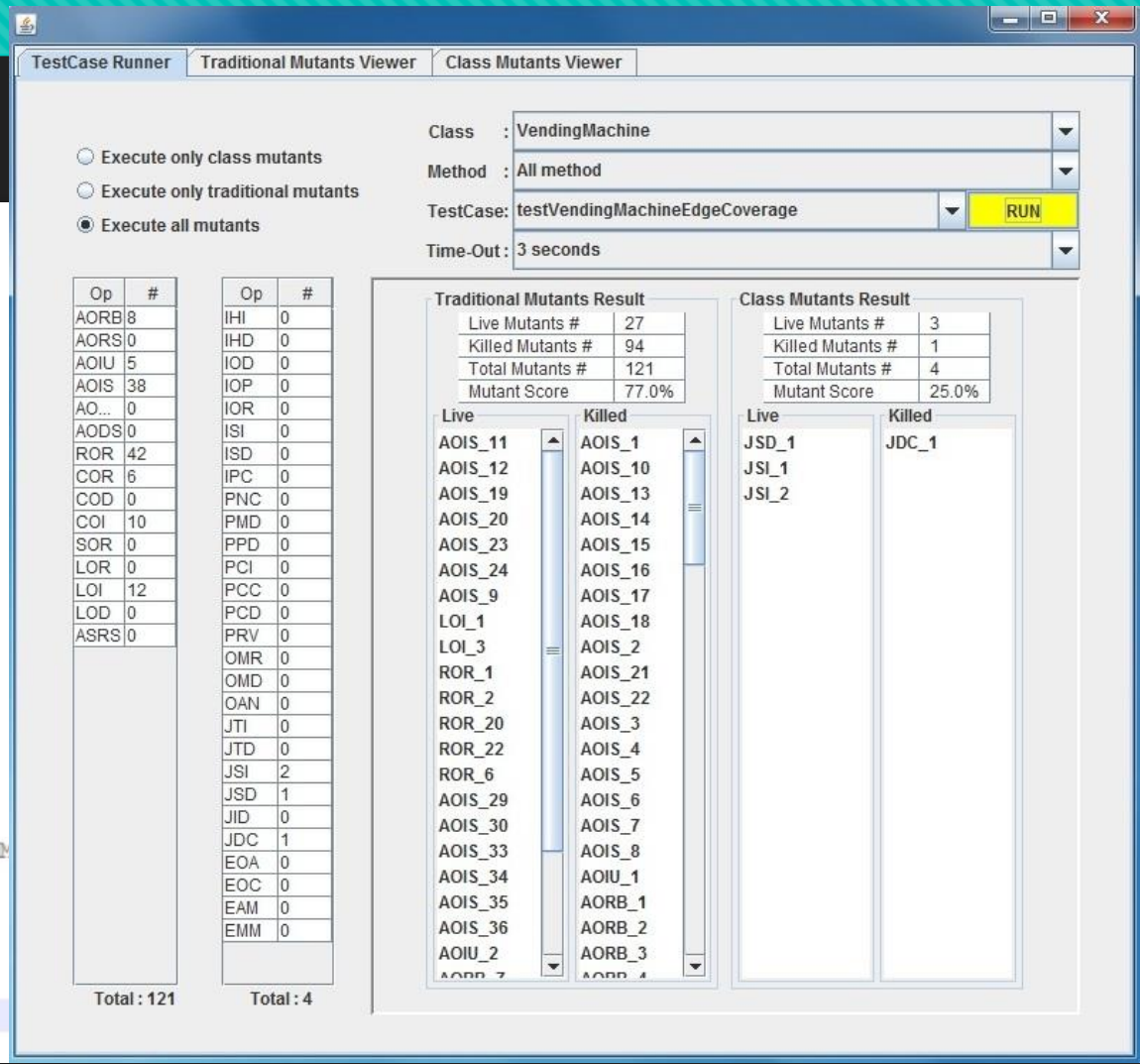




MuJava

an example of a JUnit test class for the class vendingMachine.
Its name is testVendingMachine.

```
import org.junit.After;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class testVendingMachineEdgeCoverage {
    private VendingMachine vm1;
    @Before
    public void setUp() throws Exception {
        vm1= new VendingMachine();
    }

    @After
    public void tearDown() throws Exception {
        vm1 = null;
    }

    @Test
    public void testConstructor(){
        assertEquals(0, vm1.getCredit());
        assertEquals(0, vm1.getStock().size());
    }

    @Test
    public void testCoin1(){
        vm1.coin(20);
        assertEquals(0, vm1.getCredit());
    }

    @Test
    public void testCoin2(){
        vm1.coin(25);
        vm1.coin(25);
        vm1.coin(25);
        vm1.coin(25);
        assertEquals(100, vm1.getCredit());
        vm1.coin(25);
        assertEquals(100, vm1.getCredit());
    }

    @Test
    public void testCoin3(){
        vm1.coin(25);
        assertEquals(25, vm1.getCredit());
    }

    @Test
    public void testGetChoc1(){
        StringBuffer choc = new StringBuffer().append("M
        assertEquals(0, vm1.getChoc(choc));

        vm1.coin(25);
        assertEquals(0, vm1.getChoc(choc));
    }
    ....
```



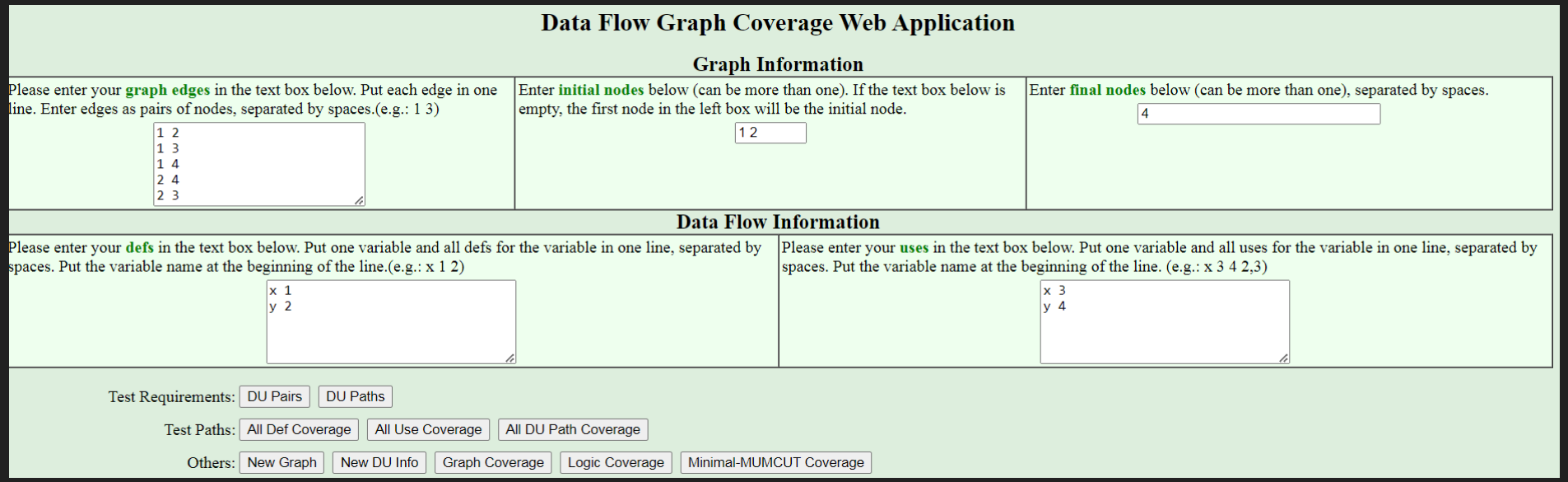
Experimental Design

Tests for edge-pair, prime path, and all-uses coverage were designed with the help of the graph coverage online web applications.

These tools provide a web-based interface that accepts definitions of graphs and defs and uses of variables.

They compute test requirements for structural criteria including prime paths and DU paths to satisfy the criteria.

The graphs were generated by hand, then submitted to the tools to create the test requirements.



Experimental Design

DU Pairs for all variables are:

Variable	DU Pairs
x	[1,3]
y	[2,4]

DU Paths for all variables are:

Variable	DU Paths
x	[1,3] [1,2,3]
y	[2,4] [2,3,4]

All Def Coverage for all variables are:

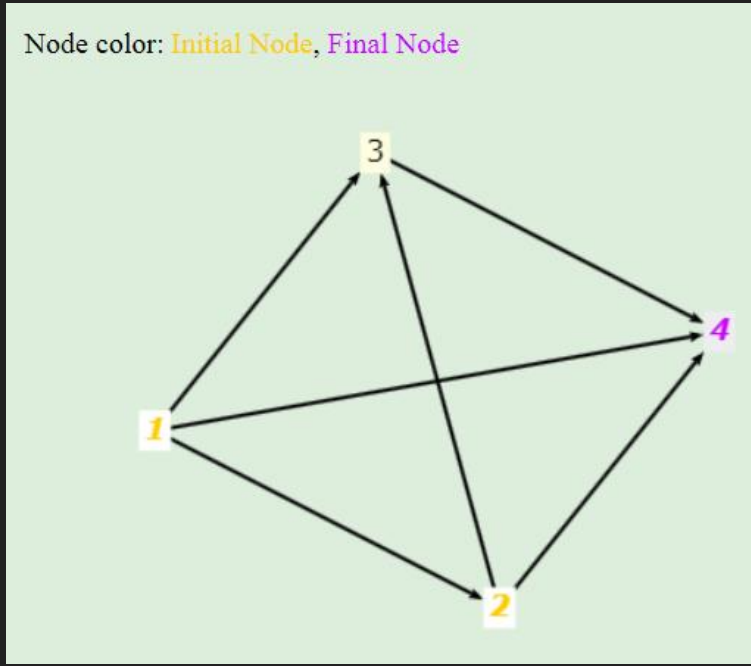
Variable	All Def Coverage
x	[1,3,4]
y	[2,4]

All Use Coverage for all variables are:

Variable	All Use Coverage
x	[1,3,4]
y	[2,4]

All DU Path Coverage for all variables are:

Variable	All DU Path Coverage
x	[1,3,4] [1,2,3,4]
y	[2,4] [2,3,4]



Experimental Design

The independent variable in this experiment is the test criterion used.

The two dependent variables:

- the number of tests.
- the number of faults found.

Twenty-nine Java classes were used.

- Since this study is strictly about unit testing, we did not seek large packages, but typical (mostly small) classes.
- They were taken from various sources(open source software websites , Java textbook)

Experimental Design

We created test sets from the same collection of values.

Our intent was to make sure the four sets of tests for each program overlapped as much as possible.

The process is:

1. we generated tests to satisfy the edge-pair criterion (*Tep*) (344 tests total).
2. Next we added 92 tests to satisfy prime paths (*Tpp*).
3. We then used the *Tpp* tests to satisfy all-uses (*Tau*), but only needed a total of 362 tests.
4. Next, we ran *Tpp* on the mutants, and eliminated tests that did not contribute to mutation coverage.
   - This left 208 tests, but some mutants were still alive.
5. Finally, we added an additional 61 tests to kill the remaining mutants (*Tmut*).

After this process, $Tpp \supset Tep$, $Tpp \supset Tau$, $|\ Tau \cap Tep\ | = 297$, and $|\ Tpp \cap Tmut\ | = 208$.

All test values were generated by hand.

Experimental Design

Each fault was seeded into a separate copy of the original Java class, thus making it obvious during execution which fault was detected.

Efficiency was measured by taking the ratio of the number of tests over the number of faults found.

The ratio estimates the number of tests required to find a fault for each test criteria.

test criteria with lower ratios can be considered to be more efficient.

Number of test requirements & tests

	Requirements	Infeasible	Tests
Edge-Pair	684	7	344
All-Uses	453	62	362
Prime Paths	849	175	436
Mutation	2919	562	269

Results

Table shows the data from the experiment, broken out by each class studied.

Column1: class number

Column2: number of seeded faults.

Column3-7: number of faults found by each criterion for each program.

Column 8-11: number of tests for each criterion.

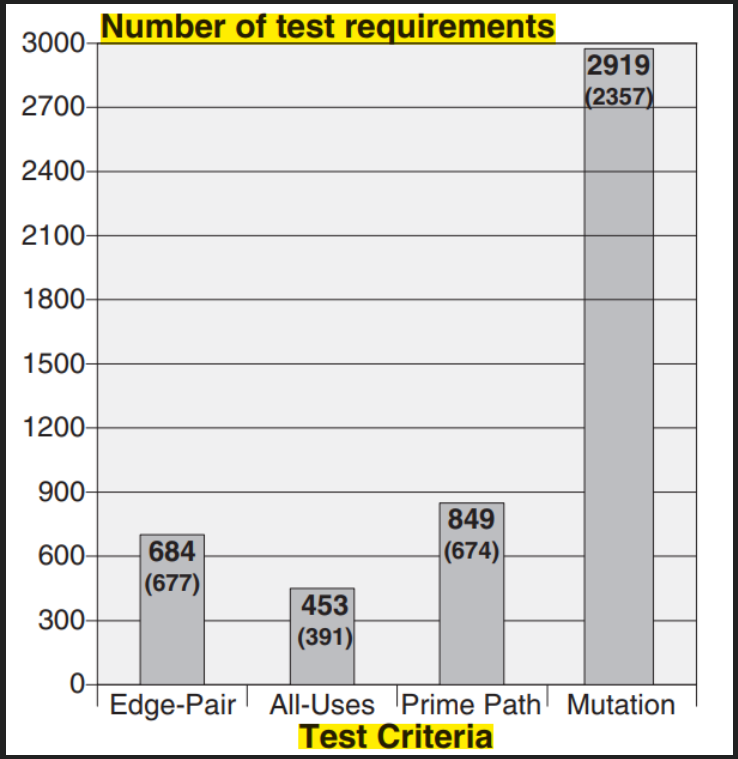
The bottom row has the totals numbers.

Experimental results

		Number of faults found				Number of test cases			
	Faults	Edge-pair	All-uses	Prime path	Mutation	Edge-pair	All-uses	Prime path	Mutation
J1	1	0	0	0	1	2	5	7	4
J2	1	1	1	1	1	3	4	5	5
J3	1	1	1	1	1	2	3	7	4
J4	1	0	0	0	0	3	4	5	4
J5	1	1	1	1	1	2	5	7	4
J6	1	0	0	0	0	2	5	7	3
J7	2	1	1	1	2	3	3	4	5
J8	1	0	0	0	1	2	2	2	5
J9	1	1	1	1	1	2	2	3	3
J10	2	1	1	2	2	5	11	12	8
J11	2	1	1	1	2	3	3	3	5
J12	6	3	3	3	3	25	25	25	9
J13	2	2	2	2	2	6	4	6	3
J14	2	2	2	2	2	9	6	9	4
J15	1	1	1	1	1	2	6	10	3
J16	19	12	12	12	16	108	106	121	42
J17	4	0	0	0	3	8	5	8	3
J18	6	2	2	2	4	18	18	18	10
J19	1	1	1	1	1	12	11	12	10
J20	1	1	1	1	1	12	11	12	10
J21	1	0	0	0	1	3	3	3	12
J22	4	1	0	1	4	11	8	11	9
J23	6	4	4	4	4	27	26	27	25
J24	4	3	3	3	4	16	15	20	21
J25	3	3	3	3	3	6	6	6	9
J26	2	2	2	2	2	10	10	14	5
J27	2	1	1	1	2	14	12	16	21
J28	4	4	4	4	4	16	21	28	11
J29	6	6	6	6	6	12	22	28	12
Totals	**88**	**55**	**54**	**56**	**75**	**344**	**362**	**436**	**269**

Results

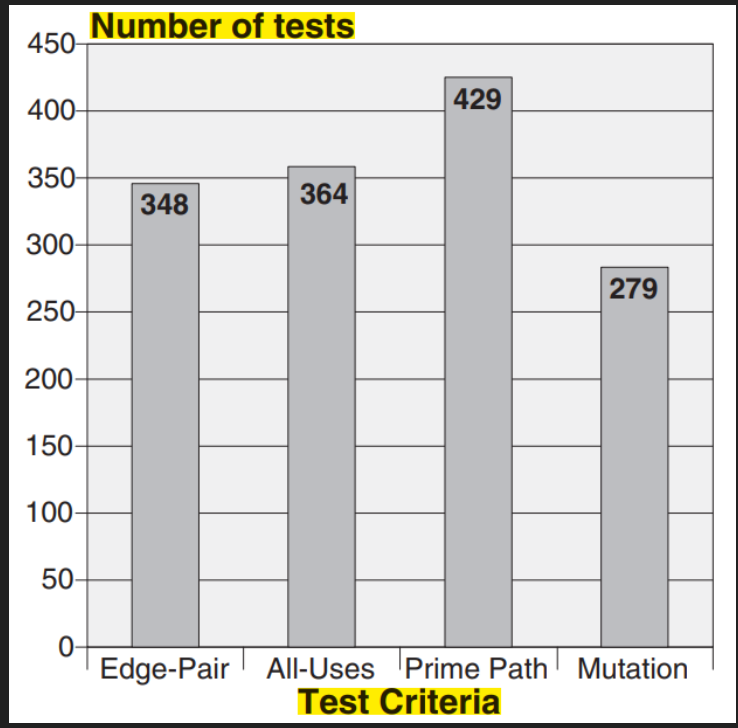
total number of test requirements for each criterion



mutation needed the fewest tests but had far more test requirements than the other criteria.

The feasible numbers of test requirements are shown in parentheses

total number of tests needed for each criterion across all subjects



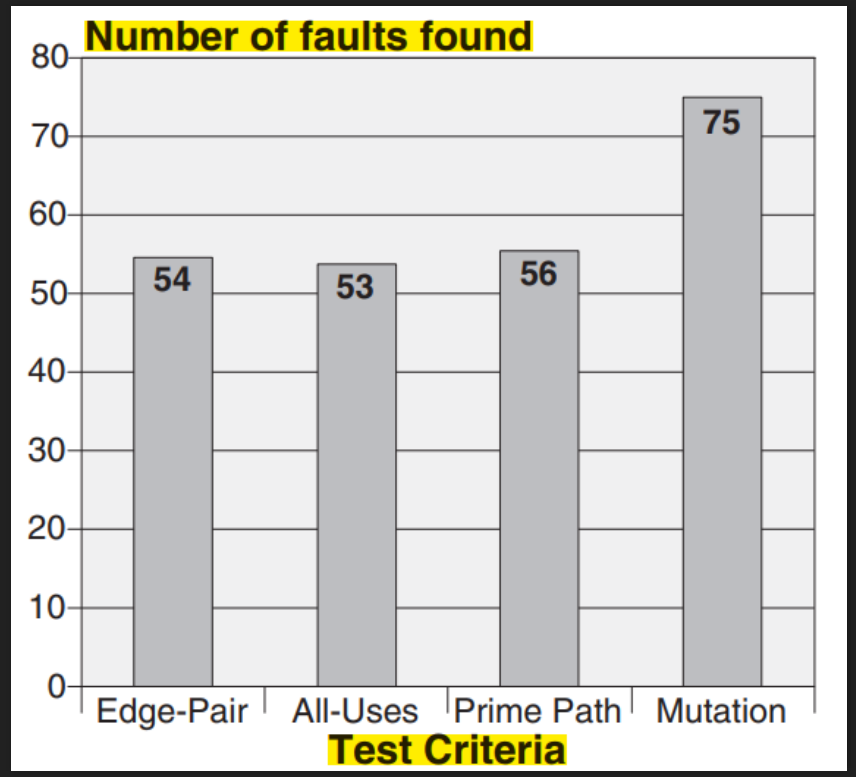
The prime path criterion needed the most tests and mutation the least.

Results

Different types of faults found

	Num	Num faults found			
Type of faults		EP	AU	PP	Mut
muJava faults	9	5	5	5	9
Mutant-like faults	19	13	13	14	17
Non mutant-like faults	60	37	36	37	49
Sum	88	55	54	56	75

Number of Faults Found by Each Criterion



Results

1.this table only counts the number of tests, not the cost of creating those tests.
2.The creation cost would vary dramatically by the amount of automation used, particularly if automatic test data generation was available.

Cost versus benefits ratio

	Tests	Faults	Cost / Benefit
Edge-pair	344	55	6.3
All-uses	362	54	6.7
Prime path	436	56	7.8
Mutation	269	75	3.6

Result
1.mutation is the most efficient criteria.
2.prime path coverage is the least efficient.

Results

our expectations were prime path coverage would found more faults than edge-pair and all-uses coverage.

However, it is surprising that the tests from all three criteria found almost the same number of faults.

We were surprised that prime path coverage required more tests than mutation coverage.

Mutation certainly has significantly more test requirements.

Which test criterion is best

The answer depends on at least 3 issues:

1. how difficult it is to compute test requirements.
2. how difficult it is to generate tests.
3. how well the tests reveal faults.

The first two, depend greatly on the level of automation used.

Computing all-use test requirements by hand is quite a bit more difficult than computing prime paths, which is in turn more difficult than computing edge-pairs.

Analysis of Specific Faults

We analyze some of the faults that were not found by some of the test sets.

The mutation tests found all of the faults that the other three criteria found, but missed the following two faults.

1.The fault is that the for loop should search from the last element to the first, but searches from the first to the last.

muJava generated 22 mutants for lastZero().

The tests {0}, {2}, {1, 0}, and {-1} killed twenty.

The fault could have been found by the test {0, 1, 0}.

expected output : 2

actual output : 0

first fault in method lastZero().

```
public static int lastZero (int[] x)
{ // if x==null throw NullPointerException
  // else return index of the LAST 0 in x
  // Return -1 if 0 is not in x
  for (int i = 0; i < x.length; i++)
  {
    if (x[i] == 0)
    {
      return i;
    }
  }
  return -1;
}
```

The correct statement should be:

“(for int i = x.length; x >= 0; i--)”

Analysis of Specific Faults

The fault is that the if statement only counts odd positive values, and misses odd negative values.

muJava generated 23 mutants for oddOrPos().

The test cases {2}, {1, 0}, and {2, 2, 2} killed twenty-one and the rest are equivalent mutants.

fault could have been found by the test case {-1}.

expected output :1

actual output : 0

The second fault is in method oddOrPos() .

```
public static int oddOrPos (int[] x)
{ // if x==null throw NullPointerException
  // else return number of elements in x
  // that are odd, positive or both
  int count = 0;
  for (int i = 0; i < x.length; i++)
  {
    if (x[i]%2 == 1 || x[i] > 0)
    {
      count++;
    }
  }
  return count;
}
```

The correct statement should be:
which counts odd negative values

```
"if (x[i]\%2 == -1 || x[i] >0),"
```

References

- https://ieeexplore.ieee.org/document/4976390
- https://www.softwaretestinghelp.com/what-is-mutation-testing/
- https://cs.gmu.edu/~offutt/mujava/
- http://cs.gmu.edu:8080/offutt/coverage/GraphCoverage