Python test generator

Paolo Tonella (Software Institute, Università della Svizzera italiana, Lugano, Switzerland)

Goal of the project

Write a search based automated test generator for Python. *The generator shall maximize condition coverage of the functions under test and will be compared against a random fuzzer used as baseline.*

1. **Write an instrumentation script that transforms the Python code under test to enable computation of the coverage fitness function [mandatory: 6/10]**
2. Develop a fuzzer that generates new test cases randomly or by mutating/ crossing over previously created tests [mandatory: 6/10]
3. Use the library Deap to define a genetic algorithm that evolves test case inputs so as to maximize condition coverage [optional: 8/10]
4. Use the tool MutPy to inject artificial faults (mutations) into the benchmark functions under test and evaluate the fault detection capability of the genetic algorithm, considering the random fuzzer as baseline [optional: 10/10]

Instrumentation

example.py

```python
def f(a: int, b: int) -> int:
    if a > 0:
        if b < 0:
            return a
    if b > 0:
        if a < 0:
            return b
    if a > b:
        return a
    else:
        return b
```

instrumentor.py

example_instrumented.py

```python
from instrumentor import evaluate_condition


def f_instrumented(a: int, b: int) -> int:
    if evaluate_condition(1, 'Gt', a, 0):
        if evaluate_condition(2, 'Lt', b, 0):
            return a
    if evaluate_condition(3, 'Gt', b, 0):
        if evaluate_condition(4, 'Lt', a, 0):
            return b
    if evaluate_condition(5, 'Gt', a, b):
        return a
    else:
        return b
```

- Define a subclass of ast.NodeTransformer and define a visit method for node type FunctionDef (to add "_instrumented" at the end of the function name) and a visit method for node type Compare (to replace a Compare expression with a Call to evaluate_condition).
- Define a function evaluate_condition(num, op, lhs, rhs) that computes the branch distance to the true and to the false branch, to be stored into some global variables, and returns True when the distance to the true branch is zero (i.e., the true branch is satisfied); False otherwise.
- The global variables containing the distances to the true/false branches are updated with the minimum between the previously stored value (if any) and the new one.
- The benchmark of functions under test to be instrumented is available inside the folder `benchmark`.

Instrumentation: implement evaluate_condition

The following relational operators should be supported by function evaluate_condition(num, op, lhs, rhs)

Types	Cmp	Op	True dist	False dist
`int`, `str` with len == 1	<	Lt	lhs - rhs + 1 if lhs >= rhs; 0 otherwise	rhs - lhs if lhs < rhs; 0 otherwise
`int`, `str` with len == 1	>	Gt	rhs - lhs + 1 if lhs <= rhs; 0 otherwise	lhs - rhs if lhs > rhs; 0 otherwise
`int`, `str` with len == 1	<=	LtE	lhs - rhs if lhs > rhs; 0 otherwise	rhs - lhs + 1 if lhs <= rhs; 0 otherwise
`int`, `str` with len == 1	>=	GtE	rhs - lhs if lhs < rhs; 0 otherwise	lhs - rhs + 1 if lhs >= rhs; 0 otherwise
`int`, `str` with len == 1	==	Eq	| lhs - rhs |	1 if lhs == rhs; 0 otherwise
`int`, `str` with len == 1	!=	NotEq	1 if lhs == rhs; 0 otherwise	| lhs - rhs |
`str` with len > 1	==	Eq	edit_distance(lhs, rhs)	1 if lhs == rhs; 0 otherwise
`str` with len > 1	!=	NotEq	1 if lhs == rhs; 0 otherwise	edit_distance(lhs, rhs)

For comparisons between string variables:

- if the strings being compared have length one, convert them to integers (e.g., via ord(s)) and apply the branch distances already defined for integers
- if the strings being compared have length greater than one, use `nltk.metrics.distance.edit_distance` as distance metric; in such a case, only equality/inequality need to be supported

Instrumentation

Conditions inside assertions or return statements should not be replaced by a call to evaluate_condition

```
def f(a: int, b: int) -> int:
    assert a > 0 and b > 0
    if a > b:
        return a > b
    else:
        return a > b
```

```
from instrumentor import evaluate_condition


def f_instrumented(a: int, b: int) -> int:
  assert a > 0 and b > 0
  if evaluate_condition(1, 'Gt', a, b):
        return a > b
  else:
        return a > b
```

Recursive calls to an instrumented function should use the new function name, with suffix "_instrumented"

```
def f(a: int, b: int) -> int:
    if a < b:
        return f(b, a)
    return a - b
```

```
from instrumentor import evaluate_condition


def f_instrumented(a: int, b: int) -> int:
    if evaluate_condition(1, 'Lt', a, b):
        return f_instrumented(b, a)
    return a - b
```

Python test generator

Paolo Tonella
Software Institute, Università della Svizzera italiana, Lugano, Switzerland

Python test generator

Paolo Tonella (Software Institute, Università della Svizzera italiana, Lugano, Switzerland)

Goal of the project

Write a search based automated test generator for Python. *The generator shall maximize condition coverage of the functions under test and will be compared against a random fuzzer used as baseline.*

1. Write an instrumentation script that transforms the Python code under test to enable computation of the coverage fitness function
2. **Develop a fuzzer that generates new test cases randomly or by mutating/crossing over previously created tests**
3. Use the library Deap to define a genetic algorithm that evolves test case inputs so as to maximize condition coverage
4. Use the tool MutPy to inject artificial faults (mutations) into the benchmark functions under test and evaluate the fault detection capability of the genetic algorithm, considering the random fuzzer as baseline

Test generation: representation and initialization

The fuzzer and the GA algorithm manipulate inputs consisting of (1) lists of `int` variables; (2) lists of `str` variables; (3) key-value pairs of type `(int, str)`. Type and number of parameters can be found in the signature of the methods under test (see folder `benchmark`), which can be assumed to have **at most 3 parameters**.

```
f_instrumented(a: int, b: int)

[1, 2]
[-1, 2]
[10, 3]
[-3, 4]
[5, -2]
[7, -12]
```

```
f_instrumented(a: str)

['']
['abc']
['er']
['1sw']
[`1-%']
[`$#$']
```

```
f_instrumented(a: str, b: int)

[('', 2)]
[('aa', 4)]
[('w$#', -3)]
[('ws', 6)]
[('gh', -2)]
[('9o97', 12)]
```

For the initialization of `int` variables, choose a random integer between MIN_INT and MAX_INT (e.g., -1000, 1000). For the initialization of `str` variables, choose a random string length between 0 and MAX_STRING_LENGTH (e.g., 10) and fill the string with random lowercase alphabetic characters (in the ASCII range [97:122]). For the initialization of key-value pairs, use respectively the random string and random integer initialisers. The initial string pool and the int pool are initialized with POOL_SIZE (e.g., 1000) random values.

Test generation: mutation

The mutation operator randomly changes any of the `int` or `str` values in the list; it changes either key or value when the individual is a key-value pair.

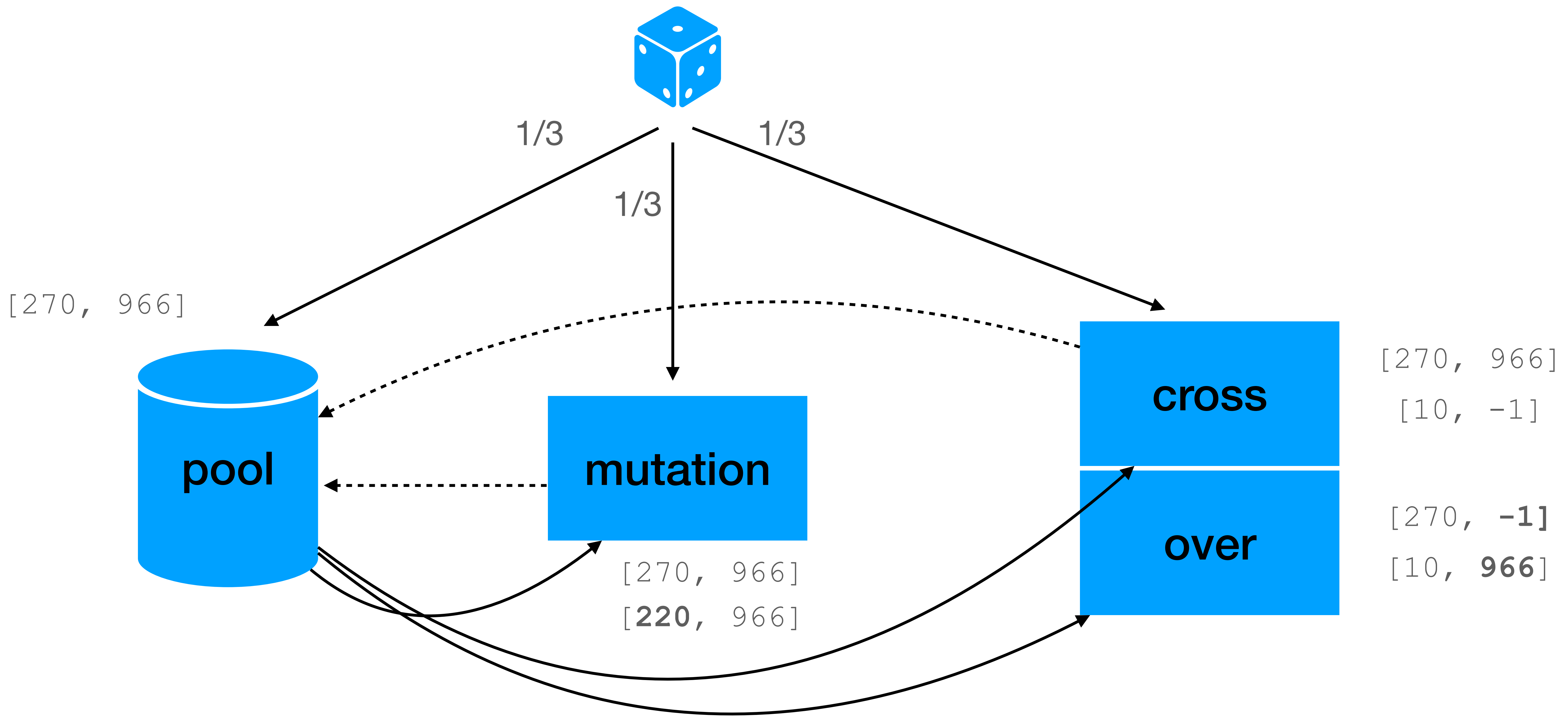
[1, 2]	[5, 2]	[‘’]	[‘’]	[(‘’, 2)]	[(‘’, 6)]
[-1, 2]	[-7, 1]	[‘abc’]	[‘atc’]	[(‘aa’, 4)]	[(‘ta’, 4)]
[10, 3]	[10, 5]	[‘er’]	[‘yr’]	[(‘w$#’, -3)]	[(‘w*#’, -3)]
[-3, 4]	[7, 4]	[‘1sw’]	[‘#sw’]	[(‘ws’, 6)]	[(‘ws’, 7)]
[5, -2]	[5, -8]	[`1-%’]	[`1%%’]	[(‘gh’, -2)]	[(‘gh’, -8)]
[7, -12]	[7, 32]	[`$#$’]	[`8#$’]	[(‘9o97’, 12)]	[(‘9p97’, 12)]

Test generation: crossover

The crossover operator randomly swaps the tails of two lists of `int`; randomly swaps the tails of two strings randomly chosen from two lists of `str`; randomly swaps the tails of the two keys of two key-value pairs

Fuzzer: test generation

New test inputs are randomly generated (with the same 1/3 probability) by: (1) using the random initialisers; (2) mutating inputs stored in the pool; (3) crossing over pairs of inputs stored in the pool. The int/str pool is initialized with POOL_SIZE random values and is then extended with any newly generated value.



Fuzzer: test execution

After dynamically loading the instrumented files (e.g., by parse/compile/exec; see sb_cgi_decode.py), to execute the function under test e.g. with two integer parameters equal to 270, 966, use: `globals()[‘f_instrumented’](270, 966)`

Upon execution collect both the input parameter values into some string variable `in` and the output value into a dictionary `out[in]`, as these values are needed for test case generation:

```
def test_f_1(self):
  y = f(270, 966)
  assert y == 966
```

value of `in = “270, 966”`

value of `out[in] = 966`

To ensure that the output value is printable into a test case oracle, make sure to escape characters with special meaning in string. For instance: `out[in] = out[in].replace(‘\\’, '\\\\').replace('"', '\\"')`

Fuzzer: generated test cases

Only test cases that increase condition coverage are kept in the archive and are reported as output test cases.
In the generated tests, the original function (e.g., f), not the instrumented one (e.g., f_instrumented) is called.

example_instrumented.py

```
from instrumentor import evaluate_condition


def f_instrumented(a: int, b: int) -> int:
    if evaluate_condition(1, 'Gt', a, 0):
        if evaluate_condition(2, 'Lt', b, 0):
            return a
    if evaluate_condition(3, 'Gt', b, 0):
        if evaluate_condition(4, 'Lt', a, 0):
            return b
    if evaluate_condition(5, 'Gt', a, b):
        return a
    else:
        return b
```

testgen_random.py

example_tests.py

```
from unittest import TestCase
from example import f

class Test_example(TestCase):
    def test_f_1(self):
        y = f(270, 966)
        assert y == 966

    def test_f_2(self):
        y = f(442, 202)
        assert y == 442

    def test_f_3(self):
        y = f(-270, -61)
        assert y == -61

    def test_f_4(self):
        y = f(-413, 414)
        assert y == 414

    def test_f_5(self):
        y = f(252, -209)
        assert y == 252
```

Python test generator

Paolo Tonella
Software Institute, Università della Svizzera italiana, Lugano, Switzerland

Università
della
Svizzera
italiana

testing AUtomated

Python test generator

Paolo Tonella (Software Institute, Università della Svizzera italiana, Lugano, Switzerland)

Goal of the project

Write a search based automated test generator for Python. *The generator shall maximize condition coverage of the functions under test and will be compared against a random fuzzer used as baseline.*

1. Write an instrumentation script that transforms the Python code under test to enable computation of the coverage fitness function
2. Develop a fuzzer that generates new test cases randomly or by mutating/crossing over previously created tests
3. **Use the library Deap to define a genetic algorithm that evolves test case inputs so as to maximize condition coverage**
4. Use the tool MutPy to inject artificial faults (mutations) into the benchmark functions under test and evaluate the fault detection capability of the genetic algorithm, considering the random fuzzer as baseline

Deap's configuration

- **Example of Deap's configuration**: sb_cgi_decode.py
- **Fitness function**:
  - sum of normalised branch distances computed only for yet uncovered branches (i.e., branches not in the archive of solutions); normalisation of branch distance d is equal to: d / (1 + d)
  - fitness to be minimized (weights = (-1.0,))
  - overall set of branches to be covered is computed during instrumentation
- **Individual**:
  - subclass of list
  - three types of individuals, corresponding to **three types of signatures**:
    - **1 to 3 int parameters**: repeat of int, with initialisation, crossover and mutate defined for list[int] individuals
    - **1 to 3 str parameters**: repeat of str, with initialisation, crossover and mutate defined for list[str] individuals
    - **a pair (str, int) of parameters**: list of one pair (str, int), with initialisation, crossover and mutate defined for individuals of type [(str, int)]
- **Test execution**:
  - as with the Fuzzer, after loading the instrumented file, call the function under test using: `globals()['f_instrumented'](270, 966)`
  - upon execution, collect inputs and output into a dictionary e.g. `out[in] = 966`
- **Test generation**:
  - Only test cases that increase coverage are kept in the archive and are reported as test case outputs
  - In the generated tests, the original function (e.g., `f)`, not the instrumented one (e.g., `f_instrumented`) is called

Python test generator

Paolo Tonella
Software Institute, Università della Svizzera italiana, Lugano, Switzerland

Python test generator

Paolo Tonella (Software Institute, Università della Svizzera italiana, Lugano, Switzerland)

Goal of the project

Write a search based automated test generator for Python. *The generator shall maximize condition coverage of the functions under test and will be compared against a random fuzzer used as baseline.*

1. Write an instrumentation script that transforms the Python code under test to enable computation of the coverage fitness function
2. Develop a fuzzer that generates new test cases randomly or by mutating/crossing over previously created tests
3. Use the library Deap to define a genetic algorithm that evolves test case inputs so as to maximize condition coverage
4. **Use the tool MutPy to inject artificial faults (mutations) into the benchmark functions under test and evaluate the fault detection capability of the genetic algorithm, considering the random fuzzer as baseline**

MutPy

https://pypi.org/project/MutPy/

```
mut.py --target example.py --unit-test example_tests.py

[*] Start mutation process:
   - targets: example.py
   - tests: example_tests.py
[*] 4 tests passed:
   - example_tests [0.00027 s]
[*] Start mutants generation and execution:
   - [#   1] COI example: [0.00626 s] survived
   - [#   2] COI example: [0.00592 s] killed by test_f_3 (example_tests.Test_example)
   - [#   3] COI example: [0.00586 s] killed by test_f_2 (example_tests.Test_example)
   - [#   4] COI example: [0.00501 s] survived
   - [#   5] COI example: [0.00592 s] killed by test_f_2 (example_tests.Test_example)
   - [#   6] CRP example: [0.00495 s] survived
   - [#   7] CRP example: [0.00558 s] survived
   - [#   8] CRP example: [0.00493 s] survived
   - [#   9] CRP example: [0.00529 s] survived
   - [#  10] ROR example: [0.00538 s] survived
   - [#  11] ROR example: [0.00508 s] survived
   - [#  12] ROR example: [0.00606 s] killed by test_f_3 (example_tests.Test_example)
   - [#  13] ROR example: [0.00523 s] survived
   - [#  14] ROR example: [0.00590 s] killed by test_f_2 (example_tests.Test_example)
   - [#  15] ROR example: [0.00513 s] survived
   - [#  16] ROR example: [0.00506 s] survived
   - [#  17] ROR example: [0.00551 s] survived
   - [#  18] ROR example: [0.00559 s] killed by test_f_2 (example_tests.Test_example)
   - [#  19] ROR example: [0.00528 s] survived
[*] Mutation score [0.17480 s]: 31.6%
   - all: 19
   - killed: 6 (31.6%)
   - survived: 13 (68.4%)
   - incompetent: 0 (0.0%)
   - timeout: 0 (0.0%)
```

Mutation score

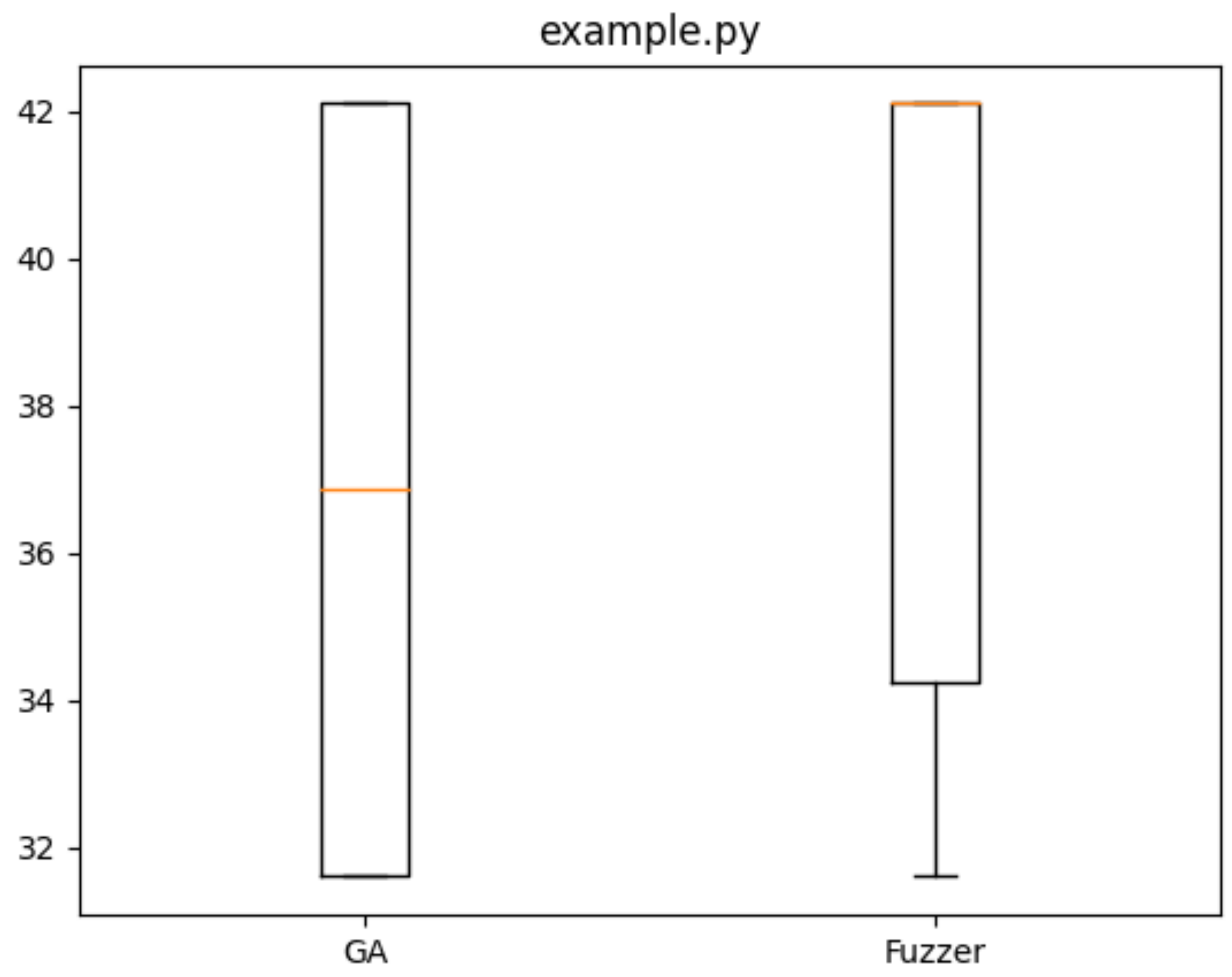
Run MutPy:

```
stream = os.popen(f'mut.py --target {py_file} --unit-test {py_test_file}')
output = stream.read()
```

Collect mutation score:

```
re.search('Mutation score \[.*\]: (\d+\.\d+)\%', output).group(1)
```

Statistical comparison



Mean mu-score:
GA = 36.85 Fuzzer = 38.95
Effect size: -0.41 (small)
Wilcoxon's p-value: 0.31

Experimental procedure:

- For each benchmark program P
  - Repeat the following experiment N times (e.g., with N = 10):
    - Generate random test cases for P using the GA generator
    - Measure the mutation score for P
    - Generate search based test cases for P using the Fuzzer
    - Measure the mutation score for P
  - Visualize the N mutations score values of Fuzzer and GA using boxplots
  - Report the average mutation score of Fuzzer and GA
  - Compute the effect size using the Cohen’s d effect size measure
  - Compare the N mutation score values of Fuzzer vs GA using the Wilcoxon statistical test

Python test generator

Paolo Tonella
Software Institute, Università della Svizzera italiana, Lugano, Switzerland