Università
della
Svizzera
italiana

Python test generator

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

Icons: flaticon.com

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 `(str, int)`. 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-%']
[`$#$']
```

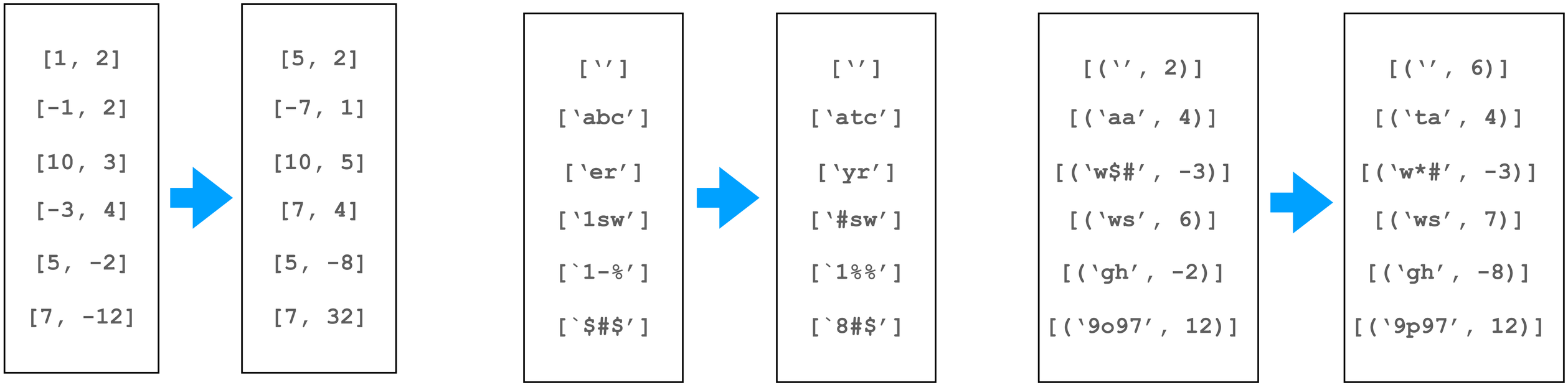
`f_instrumented(a: str, b: int)`

```
[('', 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.

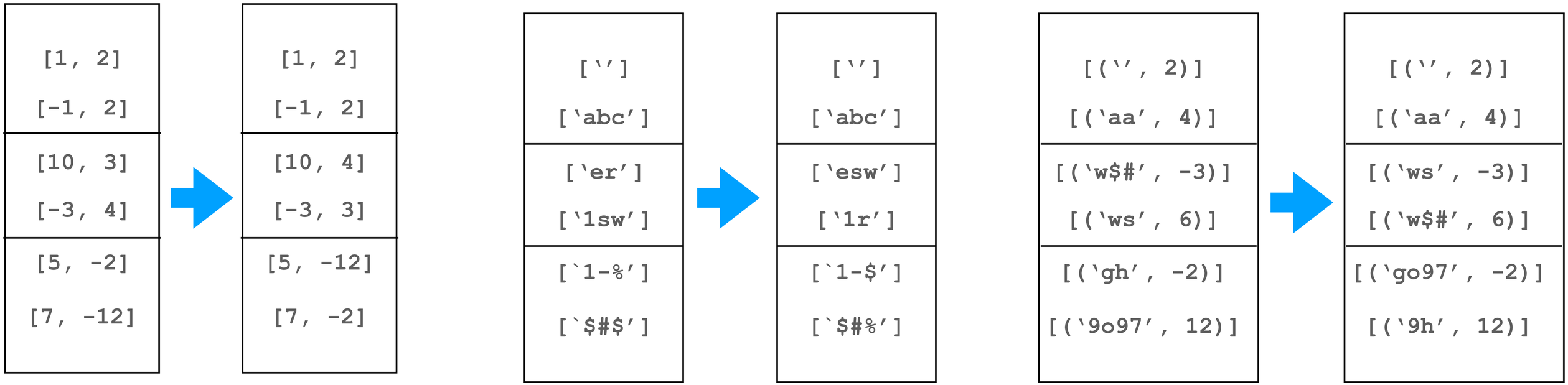
Test generation: mutation

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.



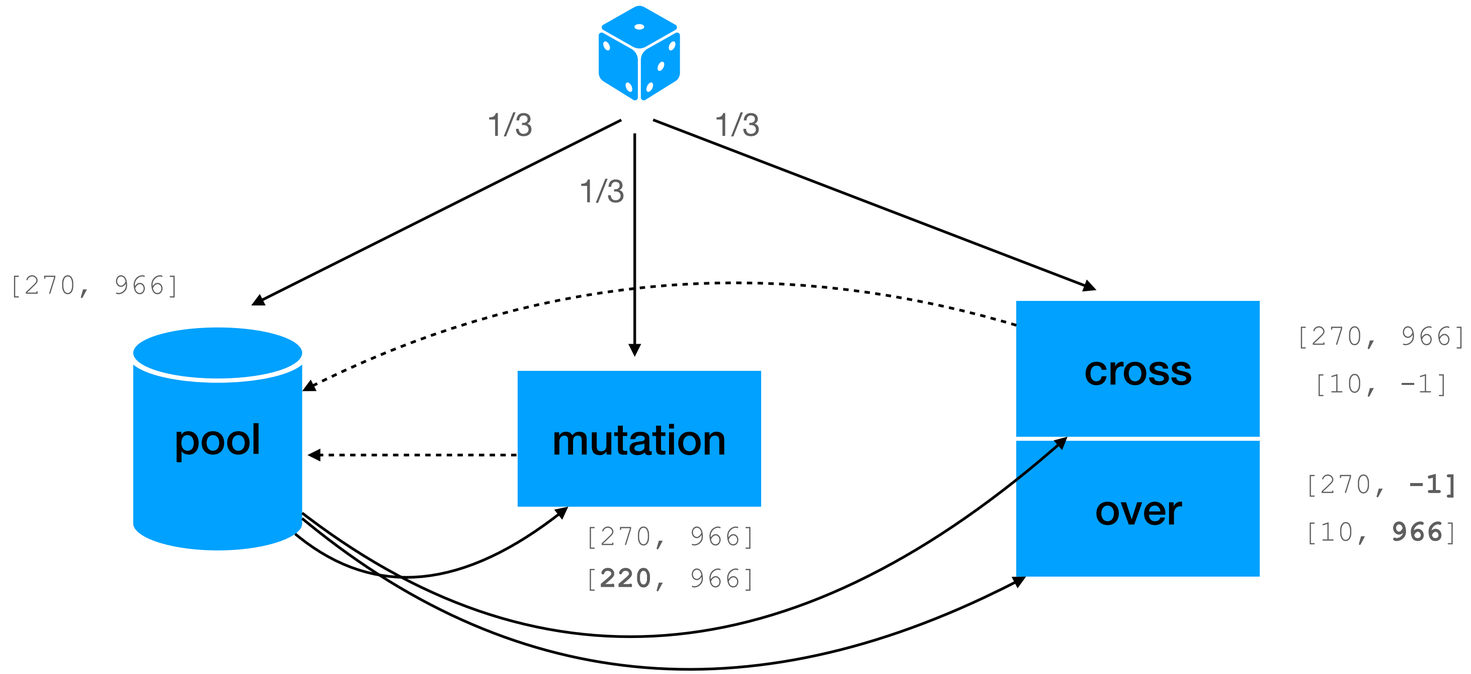
Test generation: crossover

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

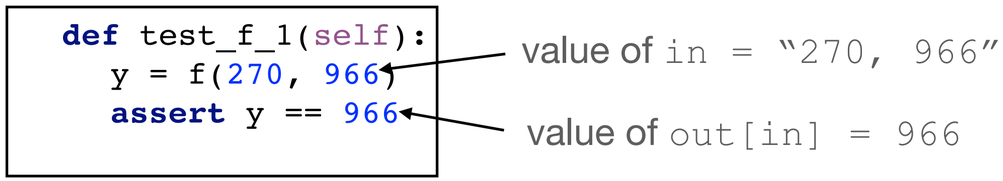
New test inputs are randomly generated (with the same 1/3 probability) by: (1) using the random initialisers; (2) mutating inputs stored in the pool; (3) crossing over pairs of inputs stored in the pool. The int/str pool is initialized with POOL_SIZE random values and is then extended with any newly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



Fuzzer: test execution

After dynamically loading the instrumented files (e.g., by parse/compile/exec; see sb_cgi_decode.py:194-196), 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:



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


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