AutoPy: An Automatic Test-Case Generation Engine for Python

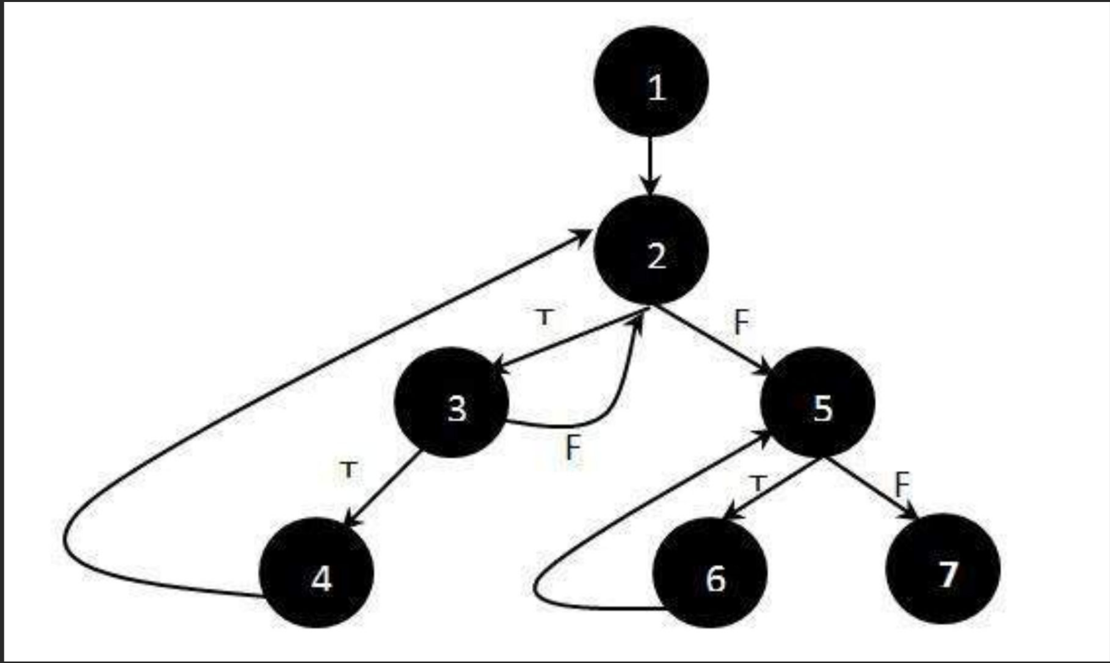
Eesha Agarwal
COS 516 (Fall 2023)

In an era where software programs form the backbone of critical infrastructure, enabling confidence in their **reliability**, **consistency**, and **dependability** is more important than ever.

"Program testing can be used to show the presence of bugs, but never their absence."

—Edsger Dijkstra

Where traditional testing falls short



Exhaustive path coverage can be an incredibly difficult benchmark to attain.

Automatic test-case generation is is a structural white-box approach to testing using static analysis, based on the technique of symbolic execution.

Automated Test-Case Generation

Symbolic Execution

+

SMT-Solving

AutoPy: A Constrained Implementation

Test-Case Generation

Analyzes execution paths to generate list of test cases

Control Flow

Implementation supports basic data types and control flow.

Unsupported Functionality

Loops, concurrency, and complex data structures not presently supported.

Automatic Test-Case Generation: An Example

```python
def two_variable_function(x, y):
    if x > 10:
        if y < 5:
            return x + y
        else:
            return x - y
    elif x == 5:
        if y > 10:
            return x * y
        else:
            return x / y
    else:
        return x ** y
```

Automatic Test-Case Generation: AST Generation

```python
def two_variable_function(x, y):
    if x > 10:
        if y < 5:
            return x + y
        else:
            return x - y
    elif x == 5:
        if y > 10:
            return x * y
        else:
            return x / y
    else:
        return x ** y
```

```
FunctionDef(name='two_variable_function', args=arguments(args=[arg(arg='x', annotation=None),
arg(arg='y', annotation=None)]))
    ├─ If(test=Compare(left=Name(id='x'), ops=[Gt()], comparators=[Num(n=10)]))
    │   ├─ If(test=Compare(left=Name(id='y'), ops=[Lt()], comparators=[Num(n=5)]))
    │   │   ├─ Return(value=BinOp(left=Name(id='x'), op=Add(), right=Name(id='y')))
    │   │   └─ Return(value=BinOp(left=Name(id='x'), op=Sub(), right=Name(id='y')))
    │   └─ Elif(test=Compare(left=Name(id='x'), ops=[Eq()], comparators=[Num(n=5)]))
    │       ├─ If(test=Compare(left=Name(id='y'), ops=[Gt()], comparators=[Num(n=10)]))
    │       │   ├─ Return(value=BinOp(left=Name(id='x'), op=Mult(), right=Name(id='y')))
    │       │   └─ Return(value=BinOp(left=Name(id='x'), op=Div(), right=Name(id='y')))
    │       └─ Else
    │           └─ Return(value=BinOp(left=Name(id='x'), op=Pow(), right=Name(id='y')))
    └─ Else
        └─ Return(value=BinOp(left=Name(id='x'), op=Pow(), right=Name(id='y')))
```

Automatic Test-Case Generation: Branching

```python
def two_variable_function(x, y):
    if x > 10:
        if y < 5:
            return x + y
        else:
            return x - y
    elif x == 5:
        if y > 10:
            return x * y
        else:
            return x / y
    else:
        return x ** y
```

```
Root
|
└ x > 10
   ├ y < 5
   |  └ Leaf
   |
   └ y >= 5
      └ Leaf
|
└ x <= 10
   ├ x == 5
   |  ├ y > 10
   |  |  └ Leaf
   |  |
   |  └ y <= 10
   |     └ Leaf
   |
   └ x != 5
      └ Leaf
```

Automatic Test-Case Generation: Output

```python
def two_variable_function(x, y):
    if x > 10:
        if y < 5:
            return x + y
        else:
            return x - y
    elif x == 5:
        if y > 10:
            return x * y
        else:
            return x / y
    else:
        return x ** y
```

```
Number of branches: 5
Branch number: 1
Branch conditions: ['x > 10', 'y < 5']
Test values: {'x': 11, 'y': 4}

Branch number: 2
Branch conditions: ['x > 10', 'not y < 5']
Test values: {'x': 11, 'y': 5}

Branch number: 3
Branch conditions: ['not x > 10', 'x == 5', 'y > 10']
Test values: {'x': 5, 'y': 11}

Branch number: 4
Branch conditions: ['not x > 10', 'x == 5', 'not y > 10']
Test values: {'x': 5, 'y': 10}

Branch number: 5
Branch conditions: ['not x > 10', 'not x == 5']
Test values: {'x': 0, 'y': 44}
```

Future Work

Boolean Support

Add support for boolean variables and operators

Dead-Code Detection

Identify unfeasible paths, i.e. dead code.

Loop Support

Implement loop support with generated or input loop invariants.

Additional Functionality

Support for try-except blocks, lists, strings, etc.

The integration of formal verification techniques like symbolic execution and constraint-solving into the realm of software-testing is a step towards a future where the robustness of programs is not just hoped for, but meticulously-crafted and confidently-assured.