Assignment 1
Automated Test Generation Fundamentals

Casper Kristiansson

January 5, 2024

1 Test Creation

If we want to write test cases that should cover all edges in the control graph which can be seen in figure 1 we can start by analyzing the possible different paths. The paths can be divided up into:

- **Test Case 1:**
  - **State:** The path list is empty
  - **Test Path:** [1, 2, 5]
- **Test Case 2:**
  - **State:** The path list contains one element but does not equal the parameter n.
  - **Test Path:** [1, 2, 3, 2, 5]
- **Test Case 3:**
  - **State:** The path list contains more than one element but none equals the parameter n.
  - **Test Path:** [1, 2, 3, ..., 2, 5]
- **Test Case 4:**
  - **State:** The path list contains elements and n is found at the first index.
  - **Test Path:** [1, 2, 3, 4]
- **Test Case 5:**
  - **State:** The path list contains elements and n exists not at the first index.
  - **Test Path:** [1, 2, 3, ..., 3, 4]

If the goal is to find possible bugs in the method we can analyze everything that happens in the method. The following inputs are examples of what should be tested to make sure that the method behaves as expected:

- Passing null as a parameter to the function
- Custom object for the parameter which possibly could have overridden the equals method

If the goal is to automatically generate tests for this function one might use a framework such as JUnit. This framework will try to automatically produce tests based on a control flow graph and by creating edge cases for the input. This means by analyzing the control flow graph it can identify possible bugs that might happen.

When trying to mimic a user's behavior it is important to test the main functionality of the function but also actions that might happen like unexpected actions. It might not always be useful to test a wide range of possible edge cases but overall it is important to test and make sure that a function works as expected.

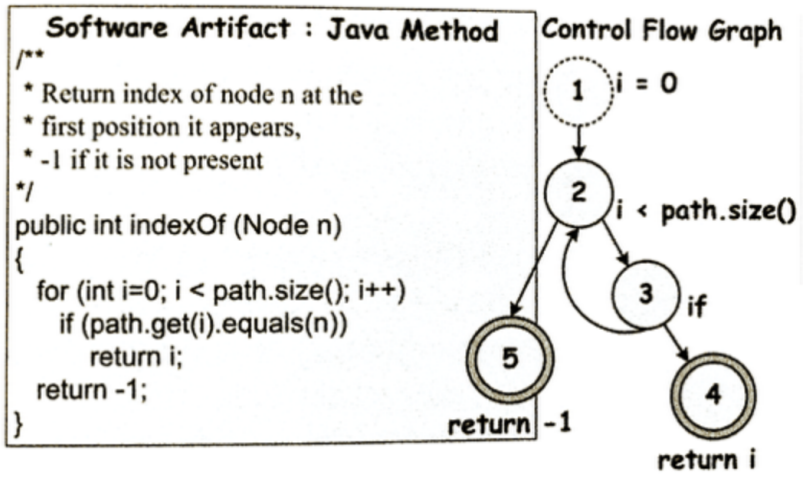

Figure 1: Example Method with Control Graph [1]

2 Taxonomy

2.1 Pynguin (Python)

1. **Software Artifact**

   (a) **Implementation**: Source, generatests test based on python source code.
   (b) **Characteristics**: Application/System because it generates tests for Python applications/systems
   (c) **Programming Language**: Python

2. **Implementation**
   (a) **Code**: Method level because the testing tools generate unit tests for different methods
   (b) **Monitoring**: Dynamic, because it executes the code and than monitor its behaviour

3. **Test Generation**
   (a) **Objective**: Code coverage, "...aims to automatically generate unit tests that maximise code coverage." [2].
   (b) **Technology**: Search-Based Testing, Chapter 3 [2].

4. **Test Execution**
   (a) **Online/Offline**: Offline

5. **Test Oracle**
   (a) **Categories**: Manual, "So far, Pynguin focuses on test-input generation and excludes the generation of oracles." [2].

2.2 Randoop (Java)

1. **Software Artifact**
   (a) **Implementation**: Source, generates tests from Java source code.
   (b) **Characteristics**: Application/System because it generates tests for Java applications/systems
   (c) **Programming Language**: Java

2. **Implementation**
   (a) **Code**: Method level because the testing tools generate unit tests for different methods
   (b) **Monitoring**: Dynamic, Randoop uses sequences "Each call in the sequence includes a method name and input arguments..." [3] to observe the execution of code to generate tests.

3. **Test Generation**
   (a) **Objective**: Code coverage

(b) **Technology**: Random, "Our technique builds inputs incrementally by randomly selecting a method call..." [3].

4. **Test Execution**

   (a) **Online/Offline**: Offline

5. **Test Oracle**

   (a) **Categories**: implicit (observer), "RANDOOP can optionally create a regression oracle for each input, ... RANDOOP guesses observer methods..." [3]

References

[1] E. P. Enoiu, "Lecture 1: Fundamentals of automated test generation," September 2019.

[2] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for python," in *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*. Springer, 2020, pp. 9–24.

[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *29th International Conference on Software Engineering (ICSE'07)*. IEEE, 2007, pp. 75–84.