

Introduction to Software Testing

Lecture 10

Automated Test Generation

Instructor: Morteza Zakeri

Slides by: **Morteza Zakeri**
in Collaboration with Course Students

March 2024

Recap: Testing Software is Hard

- If you are testing a bridge's ability to sustain weight, and you test it with 1000 tons you can infer that it will sustain weight ≤ 1000 tons
- This kind of reasoning does not work for software systems
 - Software systems are not linear nor continuous
- Exhaustively testing all possible input/output combinations is too expensive
 - the number of test cases **increase exponentially** with the number of input/output variables

Where are we?

Lecture	Topic	Tools
L01-L04	Introduction: Why test, Type of tests, Test automation, and Agile test	JUnit, NUnit
L05-L09	Functional testing (Verification), Criteria-Based Testing	Graph Coverage, Data Flow Coverage, Domain Coverage, CodA
L10	Automated test generation (Symbolic Execution, Concolic Execution, and Taint analysis, Search-based testing)	CREST, CROWN, KLEE, JDART, Triton, EvoSuite, Randoop
L11	GUI (Desktop and Web Applications) testing	Selenium, Katalon Studio, GUITAR,
L12	Performance testing (Load, Stress)	JMeter
L13	Security (penetration) testing	Burp Suite, Acunetix
L14	Negative testing (Fuzzing)	AFL, Peach, FileFuzz, DeepFuzz

What is Automated Test Generation

- Automated testing refers to the techniques which generate the test sets (test data and/or test oracles + test codes) automatically.
 - *Paper: Automated software test generation: Some challenges, solutions, and recent advances*
 - *Paper: Survey on test data generation tools*

Generating Test Cases Randomly

- If we pick **test data randomly** it is unlikely that we will pick a case where x and y have the same value
- If x and y can take 2^{32} different values, there are 2^{64} possible test cases. In 2^{32} of them x and y are equal
 - probability of picking a case where x is equal to y is 2^{-32}
- It is not a good idea to **pick the test cases randomly (with uniform distribution)** in this case
- So, **naive random testing is pretty hopeless too.**
- Various input **probability distributions** can be used.

```
bool isEqual(int x, int y)
{
    if (x == y)
        z := false;
    else
        z := false;
    return z;
}
```

Automated Test Data Generation:

Techniques, tools, and research papers

- Adaptive random testing
 - Related tool(s): ARTGen, Randoop, ...
 - Related paper(s): *A survey on adaptive random testing*
- Symbolic and concolic execution
 - Related tool(s): JDART, KLEE, [CROWN](#), ...
 - Related paper(s): *A survey of symbolic execution techniques*
- Search-based test data generation
 - Related tool(s): EvoSuite, Pynguin, ...
 - Related paper(s): *An extensive evaluation of search-based software testing: a review*

Adaptive random testing

About ART

Adaptive Random Testing (ART) is a software testing method that improves upon traditional random testing by increasing efficiency and coverage. The primary goal of ART is to reduce the number of tests required to find defects compared to conventional random testing. This method uses various algorithms and strategies to select tests in a way that uniformly covers the test space, thereby increasing the likelihood of detecting faults.

In traditional random testing methods, tests are selected completely randomly without considering the positions of previous tests.

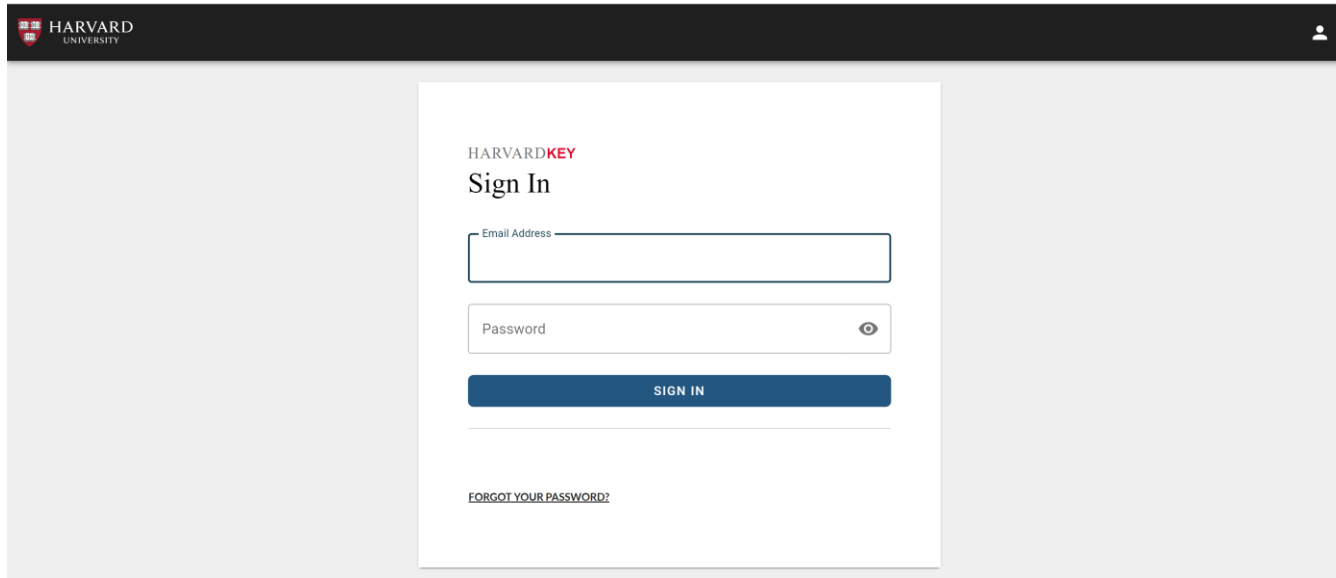
Traditional Random Testing

In Traditional Random Testing, tests are selected randomly without considering specific positions or characteristics. Typically, tests are chosen randomly from the input space without regard to the type of input or the system's previous state.

In this method, tests are often selected based on generating random numbers or other methods utilizing random events. A large number of tests are usually executed in hopes of achieving comprehensive coverage of the input space. While this approach is often fast and straightforward to implement, it may have limited effectiveness in uncovering hidden defects or weaknesses in the software.

Example

- For example, in a web application, Traditional random testing might randomly fill in input forms and click buttons without considering common user interactions or important scenarios that could occur.



The screenshot shows the Harvard University Sign In page. At the top, there is a dark header with the Harvard University logo on the left and a user profile icon on the right. Below the header, the main content area is light gray. In the center, there is a white box containing the 'HARVARDKEY Sign In' form. The form has two input fields: 'Email Address' and 'Password'. The 'Password' field has a toggle icon (an eye) to its right. Below the input fields is a blue 'SIGN IN' button. At the bottom of the form, there is a link that says 'FORGOT YOUR PASSWORD?'. The overall layout is clean and professional.

- Let's Check the codes

The key advantages of ART

- **Improved Fault Detection Capability:** By generating diverse test cases that explore different regions of the input domain, ART is more likely to uncover a wider range of defects.
- **Early Fault Detection:** The adaptive nature of ART allows it to focus on unexplored regions of the input domain, leading to the earlier detection of faults.
- **Reduced Test Suite Size:** ART can achieve the same level of fault detection with a smaller number of test cases compared to random testing, as it avoids redundant test cases.

(ART) strategies & algorithms

- Adaptive Random Testing (ART) encompasses several strategies and algorithms, each with different methods for selecting test inputs. Here are some well-known ART strategies:

(ART) strategies & algorithms

I. Feedback-Directed ART (FD-ART)

Feedback-Directed Adaptive Random Testing (FD-ART) is a hybrid testing approach that combines the principles of Adaptive Random Testing (ART) and feedback-directed test generation. The goal is to enhance the effectiveness and efficiency of the testing process by leveraging execution feedback to guide the generation of diverse and fault-revealing test cases.

(ART) strategies & algorithms

Relation to Randoop

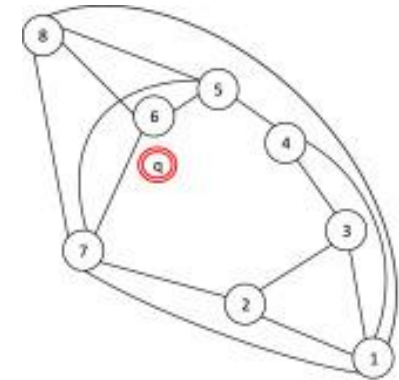
Randoop is a tool for automated test generation for Java programs that exemplifies feedback-directed test generation. It aligns with the FD-ART approach by:

- **Generating Initial Tests:** Starts with random sequences of method and constructor calls.
- **Collecting Feedback:** Executes these sequences and collects feedback on exceptions, contract violations, and other behaviors.
- **Guiding Test Generation:** Uses feedback to adaptively generate new test sequences, focusing on fault-revealing areas and ensuring test diversity.

(ART) strategies & algorithms

2. Fixed Size Candidate Set (FSCS)

In this strategy, a fixed-size set of candidate inputs is initially selected randomly. Then, the input that is the furthest away from the previously selected inputs is chosen. This process ensures a more uniform distribution of inputs across the input space.



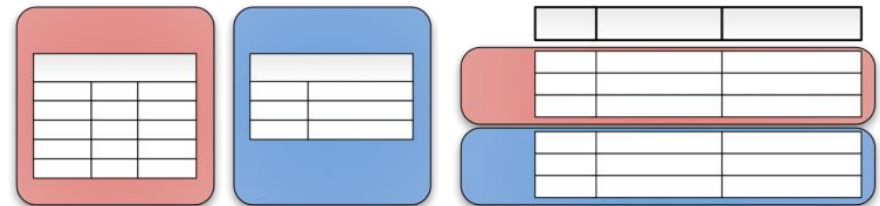
3. Distance-based ART (D-ART)

This strategy is based on the distance between inputs. In D-ART, random inputs are first selected, and then their distance from previously selected inputs is calculated. The input with the greatest distance from the others is chosen. This method also aims to increase the coverage of the input space and enhance input diversity.

(ART) strategies & algorithms

4. Grid-based ART

In this strategy, the input space is divided into a grid, and an input is selected from each cell of the grid. This approach ensures that inputs are more uniformly distributed across the input space and different areas of the space are covered.



Vertical

Horizontal

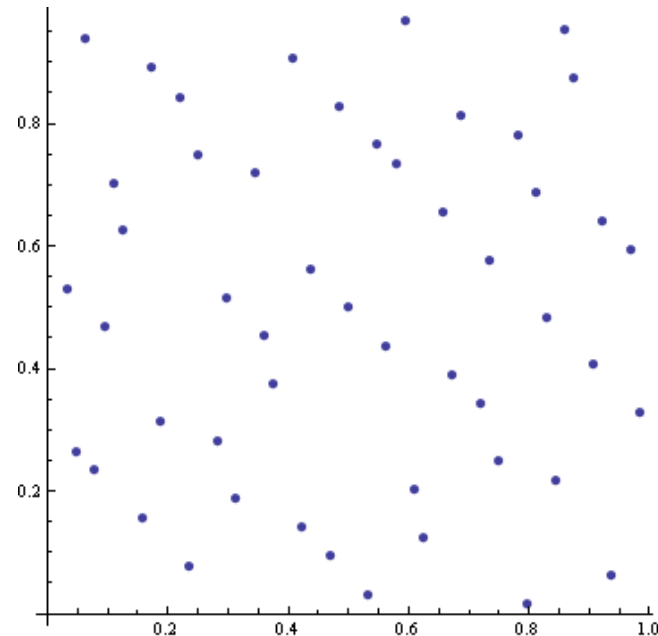
5. Partition-based ART

The input space is divided into several partitions, and then a random input is selected from each partition. This method aims to increase the coverage of the input space and prevent inputs from concentrating in a specific area.

(ART) strategies & algorithms

6. Quasi-random ART

This method uses quasi-random sequences to select inputs. Quasi-random sequences have a more uniform distribution compared to purely random sequences, making them more suitable for ART.



(ART) strategies & algorithms

7. Clustering-based ART

In this strategy, the input space is first divided into clusters, and then a random input is selected from each cluster. This method increases the diversity of inputs and improves the coverage of the input space.

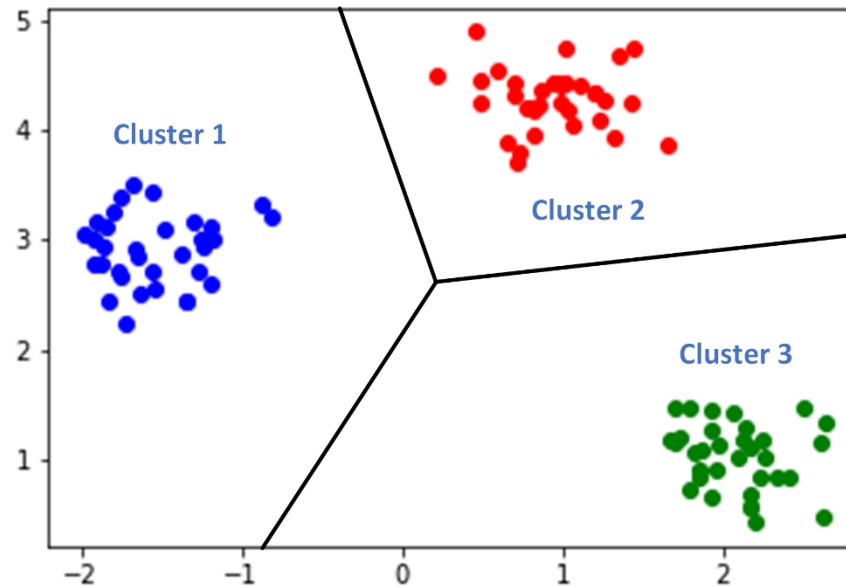
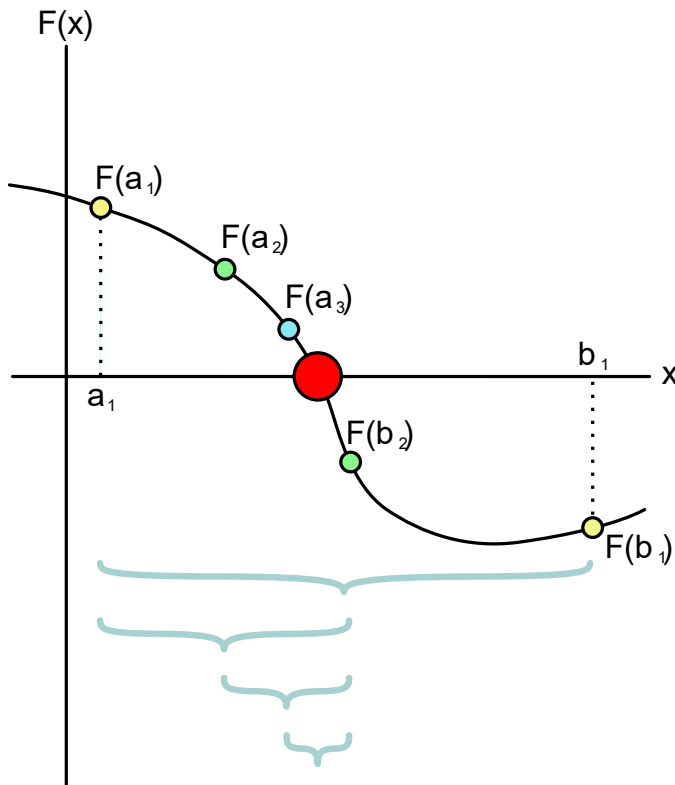


Fig.1. An Example Of Data Clustering

(ART) strategies & algorithms

8. Adaptive Random Testing by Bisection (ART-B)

This strategy uses the bisection of the input space to select inputs. The input space is repeatedly divided into smaller sections, and an input is selected from each section to increase input diversity and coverage.



Conclusion

Different ART strategies aim to improve the efficiency of random testing by more uniformly distributing and intelligently selecting inputs. The appropriate strategy depends on the type of system under test, the complexity of the input space, and the available computational resources.

Example

Let's consider an example of Adaptive Random Testing using the Fixed Size Candidate Set (FSCS) algorithm. This example will demonstrate how ART can be applied to test a software system with a two-dimensional input space.

Example: Testing a Function with a 2D Input Space

Suppose we have a function $f(x, y)$ that we want to test. The input space for f is defined as a 2D space where x and y both range from 0 to 100. Our goal is to find faults in the function by using ART.

Example

Step-by-Step Process

Step 1: Initial Random Selection

- Select the first input randomly from the entire input space.
- Let's say the first input is (20,30)

Example

Step 2: Generate Candidate Inputs

- Generate a fixed-size set of candidate inputs randomly. Suppose we decide to use 5 candidates.
- The candidate inputs might be:

(70, 80)

(10, 90)

(50, 50)

(90, 10)

(30, 70)

Example

Step 3: Calculate Distances

- Calculate the distance of each candidate input from the previously selected input(s). For simplicity, we'll use the Euclidean distance formula:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

Example

Step 3: Calculate Distances

- Distances from (20,30) to each candidate:

$$d((20, 30), (70, 80)) = \sqrt{(70 - 20)^2 + (80 - 30)^2} = \sqrt{2500 + 2500} = \sqrt{5000} \approx 70.71$$

$$d((20, 30), (10, 90)) = \sqrt{(10 - 20)^2 + (90 - 30)^2} = \sqrt{100 + 3600} = \sqrt{3700} \approx 60.83$$

$$d((20, 30), (50, 50)) = \sqrt{(50 - 20)^2 + (50 - 30)^2} = \sqrt{900 + 400} = \sqrt{1300} \approx 36.06$$

$$d((20, 30), (90, 10)) = \sqrt{(90 - 20)^2 + (10 - 30)^2} = \sqrt{4900 + 400} = \sqrt{5300} \approx 72.80$$

$$d((20, 30), (30, 70)) = \sqrt{(30 - 20)^2 + (70 - 30)^2} = \sqrt{100 + 1600} = \sqrt{1700} \approx 41.23$$

Example

Step 4: Select the Farthest Candidate

- The candidate with the greatest distance from (20,30) is (90,10) with a distance of approximately 72.80.
- Select (90,10) as the next input.

Step 5: Repeat the Process

Continue the process by generating new candidate inputs and selecting the one that is farthest from all previously selected inputs.

Example

Summary

This example demonstrates how the FSCS algorithm in ART works by iteratively selecting inputs that are farthest from previously selected ones. This helps in achieving a more uniform coverage of the input space compared to purely random testing, thereby improving the chances of finding faults.

What is **randoop**?

- Randoop is a unit test generator for Java. It automatically creates unit tests for your classes, in JUnit format.
- Randoop generates unit tests using feedback-directed random test generation. This technique pseudo-randomly, but smartly, generates sequences of method/constructor invocations for the classes under test. Randoop executes the sequences it creates, using the results of the execution to create assertions that capture the behavior of your program. Randoop creates tests from the code sequences and assertions.
- Randoop outputs two kinds of tests:
 - **error-revealing tests** that detect bugs in your current code, and
 - **regression tests** that can be used to detect future bugs.

Typical use of Randoop

- Here is a typical way to use Randoop:
 - 1.If Randoop outputs any error-revealing tests, fix the underlying defects, then re-run Randoop and repeat until Randoop outputs no error-revealing tests.
 - 2.Add the regression tests to your project's test suite.
 - 3.Run the regression tests whenever you change your project. These tests will notify you of changes to the behavior of your program.
 - 4.If any test fails, minimize the test case and then investigate the failure.
 - If a test failure indicates you have introduced a code defect, fix the defect.
 - If a test failure indicates that the test was overly brittle or specific (for example, a method's output value has changed, but the new value is as acceptable as the old value), then disregard the test.

Typical use of Randoop

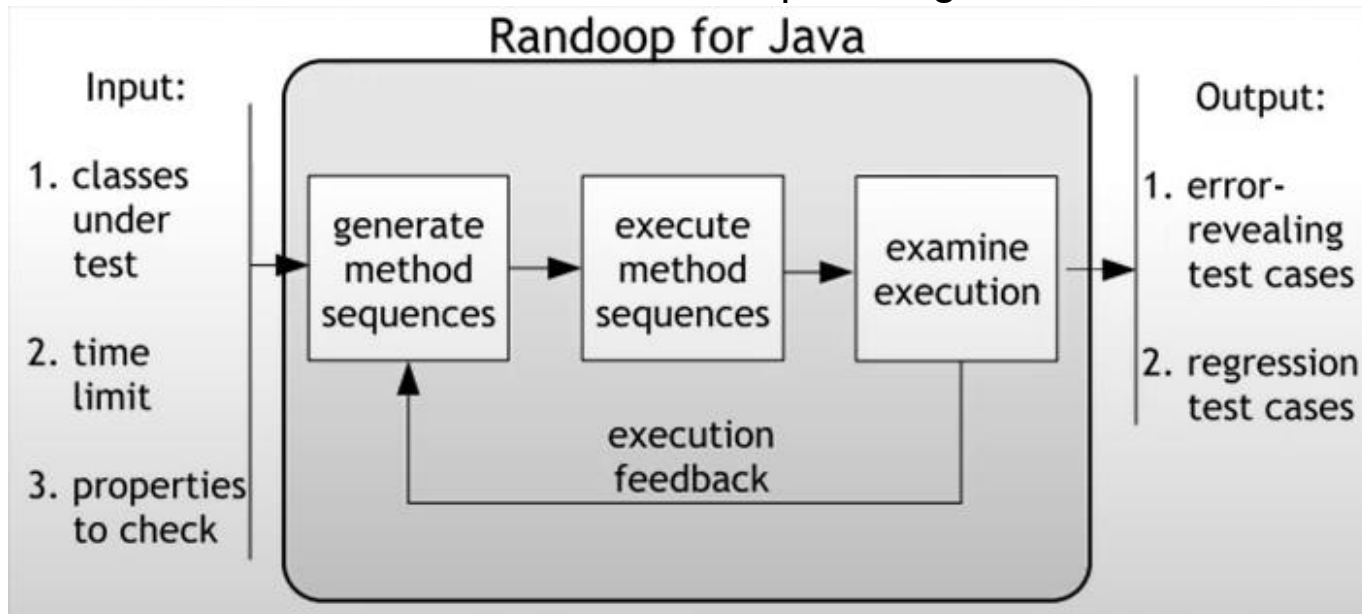
If you disregarded any test (or if you added new code that you would like to test), then re-run Randoop to generate a new regression test suite that replaces the old one.

The paper ["Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs"](#) gives additional tips for how to use Randoop-generated tests over the lifetime of a project.

A typical programmer will only ever examine very few Randoop tests - when they fail and reveal a bug or a regression failure - and then only minimized versions of them. A typical programmer will never modify Randoop tests by hand.

The idea behind randoop

- Randoop generates unit test cases using feedback_directed random testing.
- To avoid generating redundant and illegal inputs.
- Randoop creates method sequences incrementally. By randomly selecting a method call to apply and selecting arguments from previously constructed sequences.
- As soon as it is created, a new sequence is executed and checked against a set of contracts:
 - Sequences that lead to contract violations are output to the user as error revealing tests.
 - Sequences that exhibited normal behavior are output as regression tests.



Installing & Running Randoop

- Randoop runs on a Java 8 or later JVM.
- Download and unzip the file in [randoop github](#). This manual uses `${RANDOOP_PATH}` to refer to the path of the unzipped archive, and `${RANDOOP_JAR}` to refer to the location of `randoop-all-4.3.3.jar` within the unzipped archive.
- Run Randoop by invoking its main class `randoop.main.Main` :

`java randoop.main.Main command args ...`

Installing & Running Randoop

Randoop supports three commands:

gentests generates unit tests. Example use:

```
java -Xmx3000m -classpath myclasspath-- stsetneg niaM.niam.poodnar {RAJ_POODNAR}$:  
100=timil-tuptuo-- teSeerT.litu.avaj=ssalctset
```

(However, note that it is extremely unusual to use the [--testclass](#) command-line argument to specify just one class under test)

minimize minimizes a failing JUnit test suite. Example use:

```
java -cp ${RANDOOP_JAR} randoop.main.Main minimize --suitepath=ErrorTest-- avaj.0  
=htapssalcetiusmyclasspath
```


Installing & Running Randoop

help prints out a usage message. Example uses:

```
java -classpath ${RANDOOP_JAR} randoop.main.Main help
java -classpath ${RANDOOP_JAR} randoop.main.Main help gentests
java -classpath ${RANDOOP_JAR} randoop.main.Main help minimize
```

(On Windows, adjust the classpath, such as using semicolon instead of colon as the separator.)

Installing & Running Randoop

help prints out a usage message. In car builder project in bin folder:

```
java -classpath "..\lib\randoop-all-4.3.3.jar" randoop.main.Main gentests --help
```

```
PS C:\Users\Yaran\Desktop\src\refactoring_guru> cd bin
PS C:\Users\Yaran\Desktop\src\refactoring_guru\bin> java -classpath "..\lib\randoop-all-4.3.3.jar" randoop.main.Main gentests --help
ERROR: While parsing command-line arguments: unknown option name '--help' in arg '--help'
```

Code under test: which classes and members may be used by a test:

--testjar=<filename> [+]	- A jarfile, all of whose classes should be tested
--test-package=<string> [+]	- Package whose classes to test. Does not include classes in inner packages
--classlist=<filename>	- File that lists classes under test
--testclass=<string> [+]	- The binary name of a class under test
--methodlist=<filename>	- File that lists methods under test
--omit-classes=<regex> [+]	- Do not test classes that match regular expression <string>
--omit-classes-file=<filename> [+]	- File containing regular expressions for methods to omit
--omit-methods=<regex> [+]	- Do not call methods that match regular expression <string>
--omit-methods-file=<filename> [+]	- File containing regular expressions for methods to omit
--omit-field=<string> [+]	- Omit field from generated tests
--omit-field-file=<filename>	- File containing field names to omit from generated tests
--only-test-public-members=<boolean>	- Only use public members in tests [default false]
--silently-ignore-bad-class-names=<boolean>	- Ignore class names specified by user that cannot be found [default false]
--flaky-test-behavior=<enum>	- What to do if a flaky test is generated [default OUTPUT]
--nondeterministic-methods-to-output=<int>	- Number of suspected nondeterministic methods to print [default 10]

Which tests to output:

--no-error-revealing-tests=<boolean>	- Whether to output error-revealing tests [default false]
--no-regression-tests=<boolean>	- Whether to output regression tests [default false]
--no-regression-assertions=<boolean>	- Whether to include assertions in regression tests [default false]
--check-compilable=<boolean>	- Whether to check if test sequences are compilable [default true]
--require-classname-in-test=<regex>	- Classes that must occur in a test
--require-covered-classes=<filename>	- File containing class names that tests must cover

Generating tests

By default, Randoop generates and then outputs two kinds of unit tests, written to separate files.

- Error-revealing tests are tests that fail when executed, indicating a potential error in one or more classes under test.
- Regression tests are tests that pass when executed, and can be used to augment a regression test suite.

Other generated tests, classified as *invalid*, are discarded.

Generating tests

For testing Only one class with Randoop with a specific directory to write test suites and specific time limit:

```
java -classpath "..\out\production\design-patterns-java-  
main\refactoring_guru\builder\example\builders;..\lib\randoop-all-4.3.3.jar"  
randoop.main.Main gentests --testclass="builder.builders.CarBuilder" --junit-output-  
dir="..\refactoring_guru\test" --time-limit=5
```

Generating tests

For testing with a class list with a specific directory to write test suites and specific time limit:

```
java -classpath "..\out\production\design-patterns-java-main;..\lib\randoop-all-4.3.3.jar"  
randoop.main.Main gentests --classlist="..\out\myclasslist.txt" --junit-output-  
dir="..\refactoring_guru\test" --time-limit=5
```

When to use randoop

- When you have an application that has no junit test suite written for it and you need to test it.
- When you need to see know the logical mistakes in your code by generating error revealing tests.
- When you need a regression test suite to run after each change or modification in the code by you
- When you are working with TDD (test driven development) and you started with development and forget the testing

ARTGen

ARTGen is a tool designed to implement Adaptive Random Testing (ART) techniques, which aim to improve the effectiveness of random testing by distributing test cases more evenly across the input domain. ART methods, such as the Fixed-Size-Candidate-Set ART (FSCS-ART), enhance fault detection by ensuring a more uniform spread of test inputs, thereby increasing the likelihood of uncovering software defects. ARTGen leverages these principles to generate diverse and effective test cases, making it a valuable asset in software testing for identifying failure-causing inputs more efficiently.

References

- Chen, T. Y., & Merkel, R. G. (2008). "An upper bound on software testing effectiveness." *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 17(3), 16.
- Chen, T. Y., & Huang, D. T. (2013). "Adaptive Random Testing: The ART of test case diversity." CRC Press.
- Chen, T. Y., Kuo, F. C., Merkel, R. G., & Tse, T. H. (2004). "Adaptive Random Testing: The ART of test case diversity." *Journal of Systems and Software*, 83(1), 60-66.
- Chan, K. P., Cheung, S. C., & Chen, T. Y. (1996). "Proportional sampling strategy: Guidelines for generating representative failure samples." *Information and Software Technology*, 38(11), 775-782.
- Xu, X., & Parnas, D. L. (1990). "On the effectiveness of random testing." *Proceedings of the 2nd IEEE International Conference on Software Engineering*, pp. 20-26.
- [GitHub - randoop/randoop: Automatic test generation for Java](#)

Questions1

- **Question 1:**
- **What is the main objective of Adaptive Random Testing (ART)?**
- **Answer:** The main objective of Adaptive Random Testing (ART) is to improve the efficiency of random testing by ensuring a more uniform distribution of test inputs across the input space. This increases the likelihood of detecting faults by selecting inputs that are more diverse and evenly spread out.

Questions2

- Question 2:
- Name the strategies and algorithms of ART?
- Answer:
 1. Feedback-Directed ART (FD-ART)
 2. Fixed Size Candidate Set (FSCS)
 3. Distance-based ART (D-ART)
 4. Grid-based ART
 5. Partition-based ART
 6. Quasi-random ART
 7. Clustering-based ART
 8. Adaptive Random Testing by Bisection (ART-B)

Questions3

- Question 3:
- How does the Fixed Size Candidate Set (FSCS) strategy work in ART?
- **Answer:** In the Fixed Size Candidate Set (FSCS) strategy, a fixed-size set of candidate inputs is initially selected randomly. Then, the input that is the furthest away from the previously selected inputs is chosen. This process ensures a more uniform distribution of inputs across the input space.

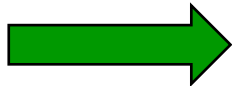
Symbolic and concolic execution

Introduction

- In computer science, symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute
- Symbolic execution explores multiple execution paths at the same time. If a program is run concretely, that is, on a specific concrete input, a single control flow path of this program is explored
- Symbolic execution is used to reason about a program path-by-path which is an advantage over reasoning about a program input-by-input as other testing paradigms use
- **Concolic** means Having elements of both concrete (normal) and symbolic processing

Symbolic Execution


- If ((pres<pres_min) || (pres>pres_max)){
...
} else{
...
}
- pres = 460; pres_min=640; pres_max=960



If ((pres<pres_min) || (pres>pres_max)){
...
} else{
...
}

Symbolic Execution


- If ((pres<pres_min) || (pres>pres_max)){
...
} else{
...
}
- pres = Sym; pres_min=MIN; pres_max=MAX
- [Path condition I: Sym<MIN]



```
If ((pres<pres_min) || (pres>pres_max)){  
...  
} else{  
...  
}
```

Symbolic Execution

- If ((pres<pres_min) || (pres>pres_max)){
...
} else{
...
}
- pres = Sym; pres_min=MIN; pres_max=MAX
- [Path condition 2: Sym>MAX]

 If ((pres<pres_min) || (pres>pres_max)){
...
} else{
...
}

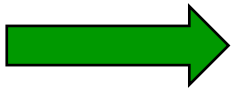
Symbolic Execution

- If ((pres<pres_min) || (pres>pres_max)){
...
} else{
...
}
- pres = Sym; pres_min=MIN; pres_max=MAX
- [Path condition 3: Sym>=MIN && Sym<=MAX]

If ((pres<pres_min) || (pres>pres_max)){

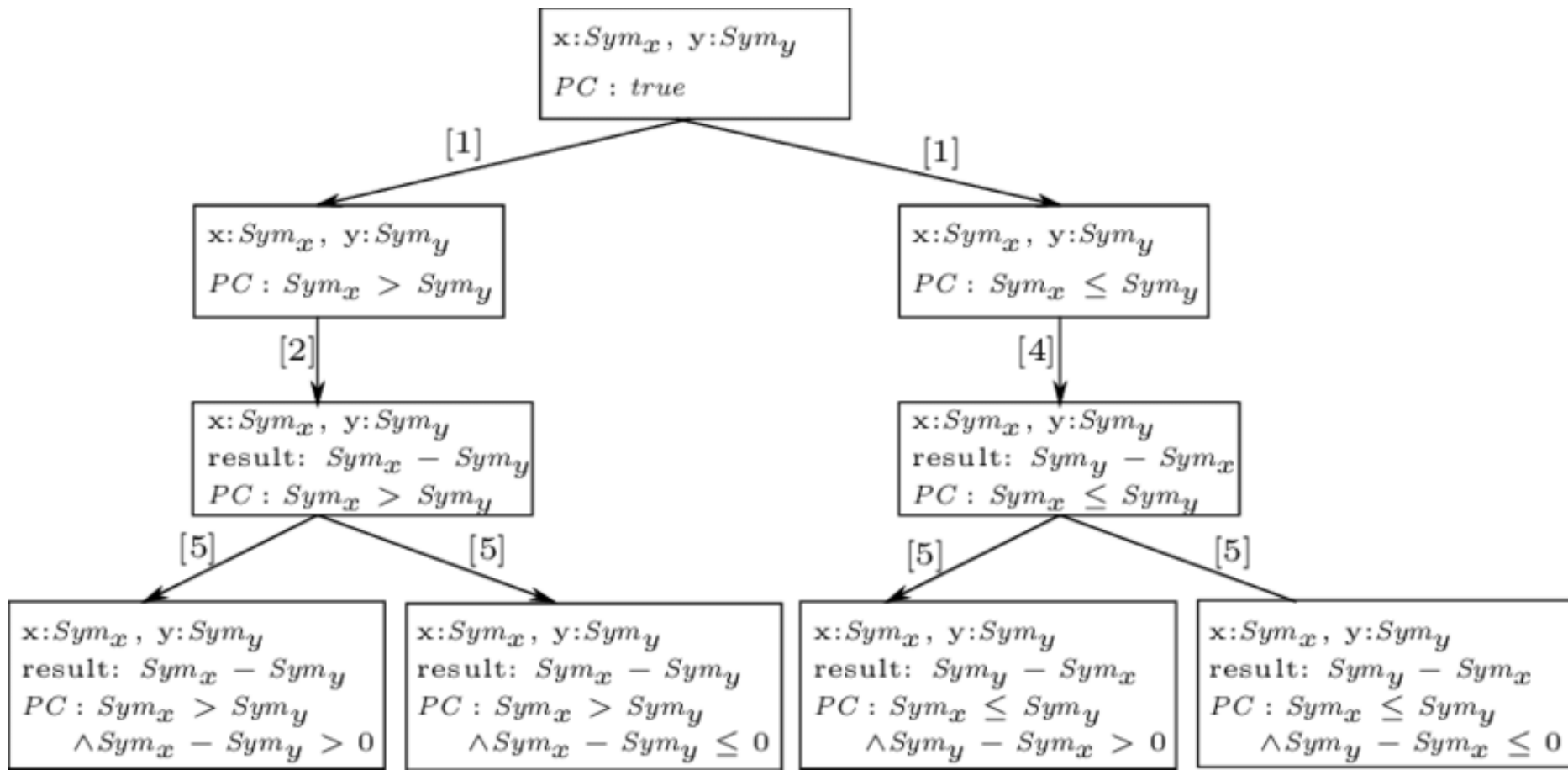
...
} else{

...
}



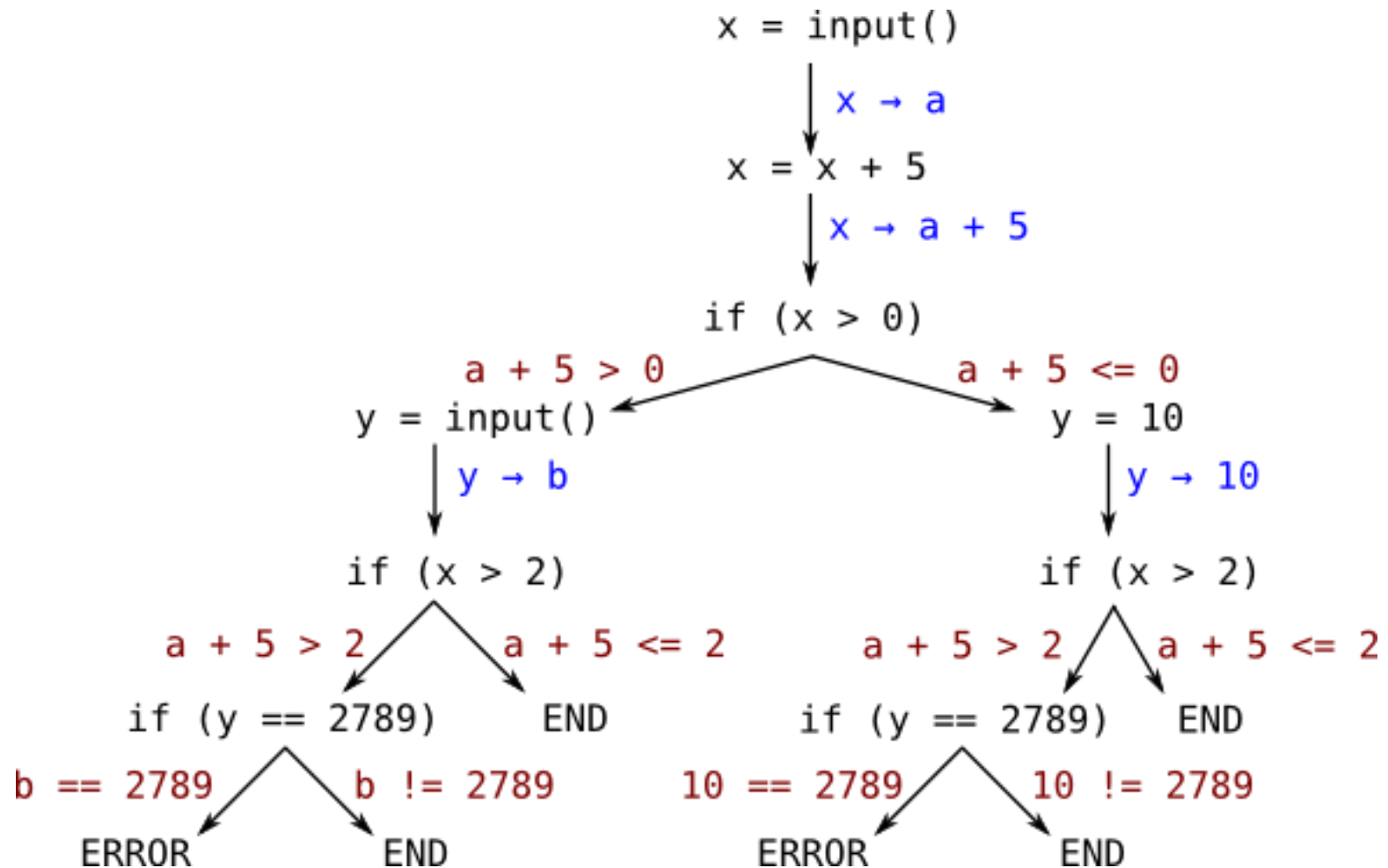
Symbolic Execution

- Symbolic Execution Tree example:



Concolic Execution

- Concolic Execution Tree example:



Difference of CFG and SET

Source Code	CFG	Symbolic Execution Tree
<pre> 1. int mytest(int x, int y){ 2. if(x>0) 3. if(x+y>10) 4. return 1; 5. else 6. return 0; 7. }</pre>	<pre> graph TD enter[enter: int test (int x,int y)] --> block1[block1:if(x>0)] block1 -- "x <= 0" --> exit[exit] block1 --> block2[block2:if(x+y>10)] block2 -- "x+y > 10" --> block3[block3:return 1] block2 -- "x+y <= 10" --> block4[block4:return 0] block3 --> exit block4 --> exit</pre>	<pre> graph TD root["x:X y:Y"] --> node1["x:X y:Y pc:X>0"] root --> node2["x:X y:Y pc:X<=0"] node1 --> leaf1["x:X y:Y pc:(X>0)∩(X+Y>10)"] node1 --> leaf2["x:X y:Y pc:(X>0)∩(X+Y<=10)"] node2 --> exit[]</pre>

Symbolic Execution

- Analysis of programs with unspecified inputs
 - Execute a program on symbolic inputs
- Symbolic states represent sets of concrete states
- For each path, build a path condition
 - Conditions on inputs: for the execution to follow that path
 - Check path condition satisfiability: explore only feasible paths
- Symbolic state
 - Symbolic values/expressions for variables
 - Path conditions
 - Program counter

not to use Symbolic execution!

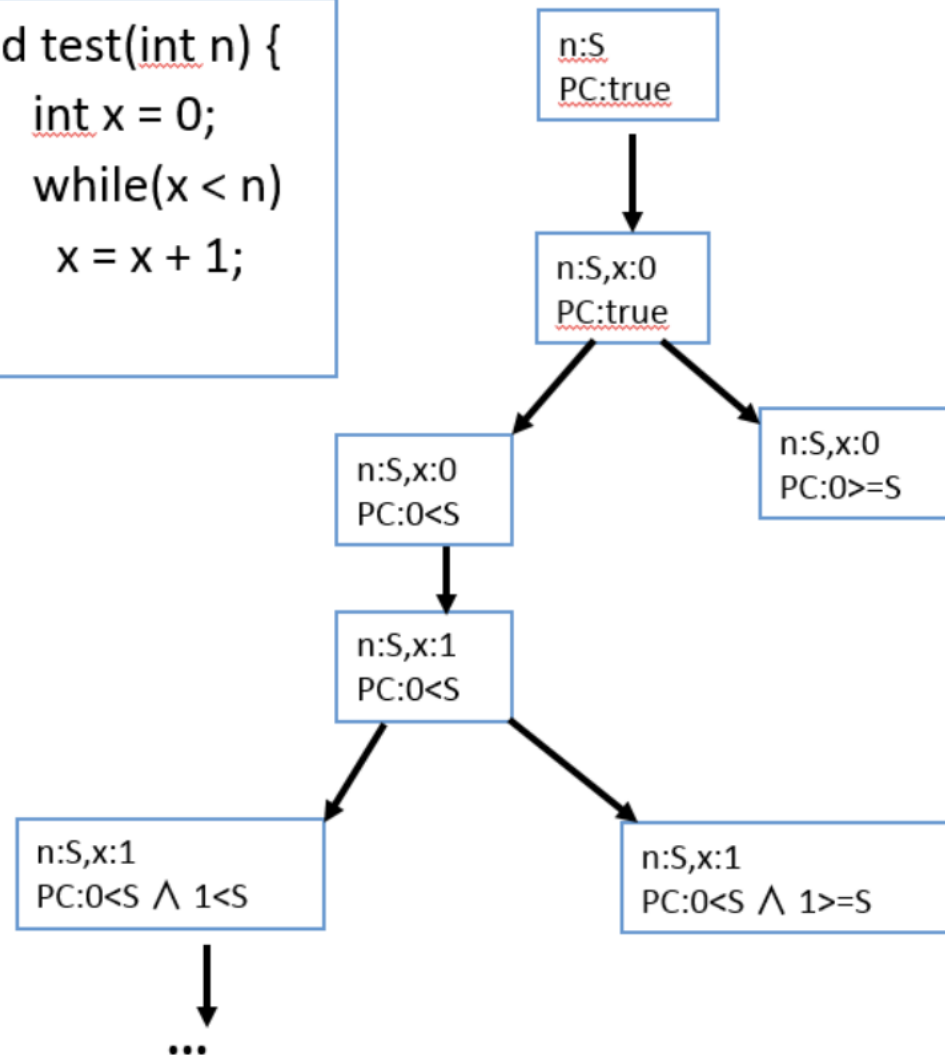
- Performing symbolic execution on looping programs
 - May result in an infinite execution tree
 - Perform search with limited depth to avoid an infinite execution tree
- When you have many merge points – conditional where the test-expression itself is symbolic
 - You end up with the usual combinatorial explosion and your SAT/SMT solver will struggle to handle all the combinations
 - In such cases people usually prefer to use “directed automated random testing” in combination with symbolic testing instead

not to use Symbolic execution!

Example Code

```
void test(int n) {  
    int x = 0;  
    while(x < n)  
        x = x + 1;  
}
```

Infinite symbolic execution tree



What is Klee

- KLEE is a dynamic symbolic execution engine designed to analyze and test software by executing it with symbolic inputs
- KLEE operates on LLVM bitcode, allowing it to analyze a wide range of software

executes the program symbolically

generates path constraints

solves the constraints

Execute and find errors

KLEE Goal

1. hit every line of executable code in the program
2. detect any potential input values that could lead to errors during risky operations such as dereferencing, assertions, segmentation, division

Usage

1. Compiling to LLVM bitcode



```
1 $ clang -I -emit-llvm -c -g -O0 -Xclang -disable-O0-optnone get_sign.c
```

`clang -emit-llvm`: compile the app to llvm bitcode

`-I`: used so that the compiler can find **<klee/klee.h>**

`-g`: to add debug information to the bitcode file

`-O0 -Xclang -disable-O0-optnone`: KLEE optimization

Usage

2. Running KLEE



```
1 $ klee get_sign.bc
```

Usage

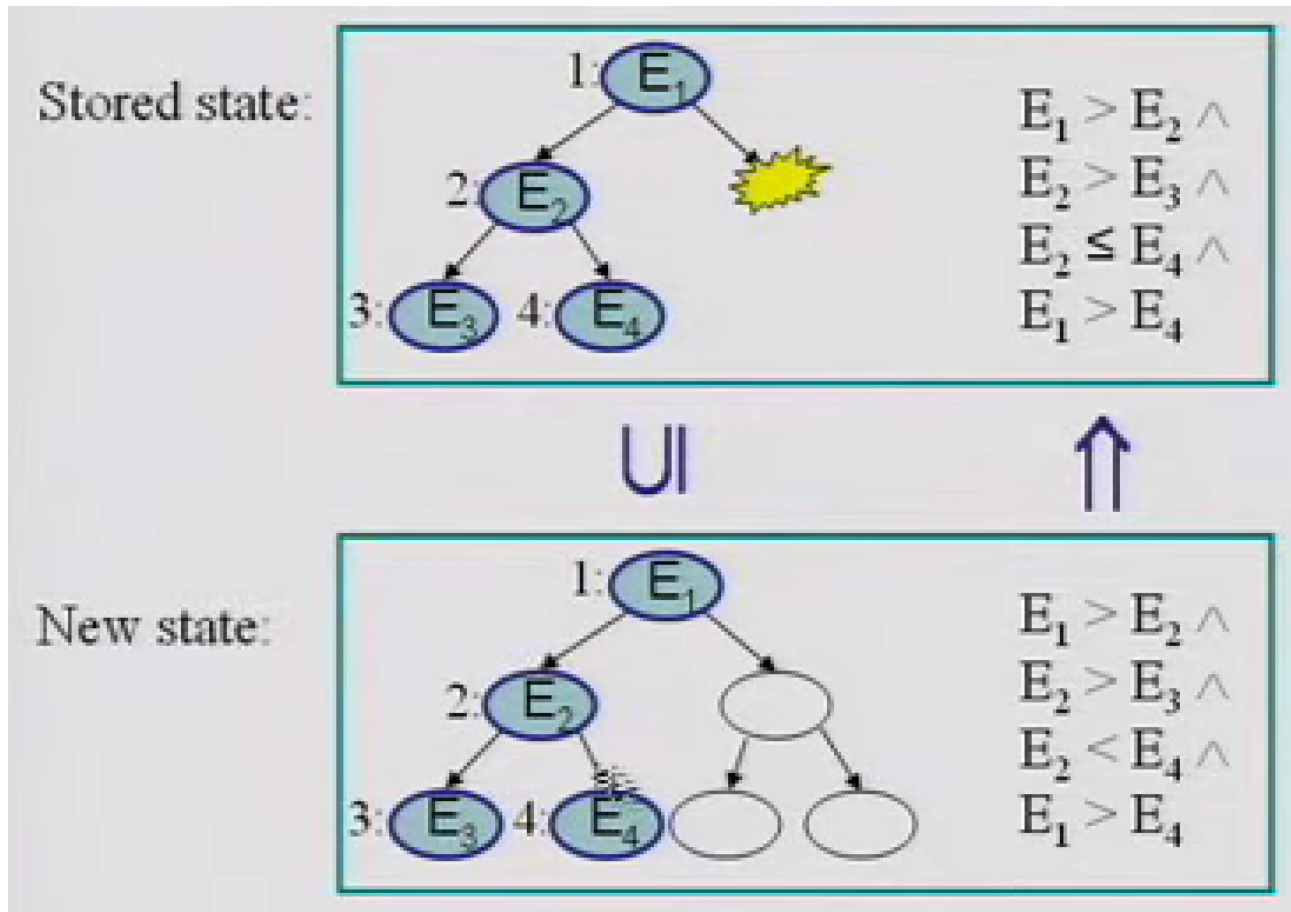
3. output



```
1 KLEE: output directory = "klee-out-0"  
2  
3 KLEE: done: total instructions = 33  
4 KLEE: done: completed paths = 3  
5 KLEE: done: partially completed paths = 0  
6 KLEE: done: generated tests = 3
```

How to improve state numbers?

- State Matching: Subsumption Checking
 - Obtained through DFS traversal of “rooted” heap configurations
 - Roots are program variables pointing to the heap



How to improve state numbers?

- Abstract subsumption
 - Abstraction
 - Store abstract versions of explored symbolic states
 - Subsumption checking to determine if an abstract state is re-visited
 - Decide if the search should continue or backtrack
 - Preserves errors to safety properties/useful for testing

How to improve state numbers?

- Shape abstraction for linked lists
 - Summarize contiguous list elements not pointed to by program variables into summary nodes
 - Valuation of summary node
 - Union of valuations of summarized nodes
 - Subsumption checking between abstracted states
 - Treat summary node as an ordinary node
- Abstraction for arrays
 - Represent array as a linked list
 - Abstraction similar to shape abstraction for linked lists

Key challenges and solutions

- **Lifting to Intermediate Representation (IR):** To address the complexity of native instructions and support multiple architectures, symbolic executors transform native instructions into an IR, like VEX or LLVM-IR, which simplifies program analysis.
- **Control Flow Graph (CFG) Reconstruction:** There are techniques for reconstructing CFGs from binaries, which is challenging due to indirect jumps and the dynamic nature of binary execution.
- **Code Obfuscation:** The impact of code obfuscation on symbolic execution, noting that obfuscation is used to hinder analysis and can lead to imprecision or excessive resource usage.

CFG Reconstruction

- **Challenges in CFG Reconstruction:**
 - Indirect Jumps: The targets of indirect jumps are determined at runtime, making them difficult to predict statically.
 - Iterative Refinement: CFG reconstruction is an iterative process that refines the graph using various program analysis techniques.
- **Techniques Used:**
 - Value-Set Analysis (VSA): This technique over-approximates certain program state properties, like possible targets of an indirect jump.
 - Initial CFG Generation: An initial CFG is created with special nodes for unresolved indirect jump targets, akin to widening a fact to the bottom of a lattice in dataflow analysis.
 - On-Demand VSA: When more precise information is required, VSA is applied to refine the CFG.

Code Obfuscation

- **Challenges for Analysis:**

- Obfuscated code poses significant challenges for symbolic/concolic execution in program analysis due to the introduction of complexities that lead to imprecision or excessive resource usage.

- **Techniques:**

- Conditional Branch Transformation: Obfuscation tools may convert conditional branches into indirect jumps, complicating symbolic analysis.
- Runtime Code Self-Modification: This technique can hide conditional jumps based on symbolic values, making them difficult to detect during analysis.
- Use of Mathematical Conjectures: Some obfuscation techniques involve inserting loops based on unsolved mathematical conjectures, which eventually converge to a known value and are combined with the original branch condition.

Questions

1. **What is Concolic execution? briefly explain each part of it.**

Concolic execution is a type of execution that uses both Concrete and Symbolic execution methods

- Concrete execution is the normal method of executing a program with actual input values

- In computer science, symbolic execution is a means of analyzing a program using symbolic values to determine what inputs cause each part of a program to execute

Symbolic execution explores multiple execution paths at the same time.

Questions

2. In which cases should we avoid using symbolic execution?

- When you have many merge points – conditional where the test-expression itself is symbolic – You end up with the usual combinatorial explosion and your SAT/SMT solver will struggle to handle all the combinations
- Performing symbolic execution on looping programs may result in an infinite execution tree

Questions

3. What are the difference of CFG and SET?

- Control Flow Graph (CFG) represents a program's structure with nodes as basic blocks of code and edges as control flows between them. It analyzes and optimizes program flow by evaluating conditions within each block, independent of specific input values.
- A Symbolic Execution Tree (SET) represents execution paths with nodes for program states and edges for symbolic conditions. It explores all possible paths, finds bugs, and generates test cases, determining constraint paths from the start to the current block. The tree may fork on dangerous operations to analyze different outcomes.

References

- [1] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A Survey of Symbolic Execution Techniques," *ACM Computing Surveys*, vol. 51, no. 3, pp. 1–39, Jul. 2018
- [2] Khurshid, Sarfraz and Pasareanu, Corina and Visser, Willem, "Generalized Symbolic Execution for Model Checking and Testing", Jul.2003
- [3] Cristian Cadar, D. Dunbar, and D. Engler, "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs," pp. 209–224, Dec. 2008

Search-based software testing (SBST)

Search-Based Software Testing (SBST)

- Search-Based Software Testing is the use of a meta-heuristic optimizing search technique, such as a Genetic Algorithm, to automate or partially automate a testing task
 - for example the automatic generation of test data
- Key to the optimization process is a **problem-specific** fitness function
- The role of the fitness function is to guide the search to good solutions from a potentially infinite search space, within a practical time limit

Search-Based Software Testing (SBST)

- **The most important part:** The conversion of test criteria to objective functions is required for test data generation
- Objective tasks compare and contrast results of the search with respect to the all search goal lines

Genetic Algorithm

- Genetic algorithm is a heuristic method, one of the types of the evolutionary algorithms
- Its goal is not to find the optimal and best solution, but to find one that is close enough to it
- A genetic algorithm is considered completed if a certain number of iterations are, or if the satisfactory value of the fitness function was obtained
- Generally, the genetic algorithm solves the problem of maximizing or minimizing, and the adequacy of each decision (chromosome) is assessed using the fitness function

Genetic Algorithm

- GA works according by the following principle:
- **Initializing:** Establishing fitness function. Forming the initial population. The initial population is created by random filling of genes in the chromosomes
- **Evaluation of population:** Each of the chromosomes is evaluated by the fitness function
- **Selection:** After each chromosome obtain its own value, the selection of the best chromosomes takes place. Selection can be done by different methods
 - For example, take the first n chromosomes sorted by value of the fitness function, or only chromosomes with maximum value of the fitness function, etc

Genetic Algorithm

- **Crossover:** After selection and retrieving the suitable chromosomes to solve the problem, they crossover with each other. Crossover occurs based on the selection of a specific position in the two chromosomes (crossover point) and mutual replacement of parts.
- **Mutation:** In random order, a random gene can change values to a random one
 - The main goal of mutation is to obtain solutions that could not be produced with existing genes. This will allow, firstly, to avoid falling into local extremes, since the mutation may allow the algorithm to go a completely different path, and secondly, to “dilute” the population in order to avoid a situation where there are only identical chromosomes in the entire population that will not move towards a global solution

Genetic Algorithm

- After all stages of the genetic algorithm have been completed, it is estimated whether the population has reached the desired accuracy of the solutions, or whether a certain number of populations have been reached. If these conditions have been met, the algorithm stops working. Otherwise, the cycle is repeated with the new population until the conditions are reached

Assigning Weights

- Search for a single path in the program code works as follows:
- The first operation is assigned a weight, for example, 100.
- Each subsequent operation is also assigned a weight – if there are no conditions or cycles, the weight is equal to the weight of the previous operation
- Weight of the condition is assigned in accordance with the following rule. If the condition contains only one branch (only if ...), then the weight of each operation is reduced on 0.8. If the condition is divided into several branches (if ... else ...), then the weight is divided into equivalent parts - for two branches 50 / 50, for three 33 / 33 / 33, etc
- Weights of operations in the cycle remain the same, but can also be multiplied by a certain weight, if it is necessary to increase the significance of the cycles during testing

Assigning Weights

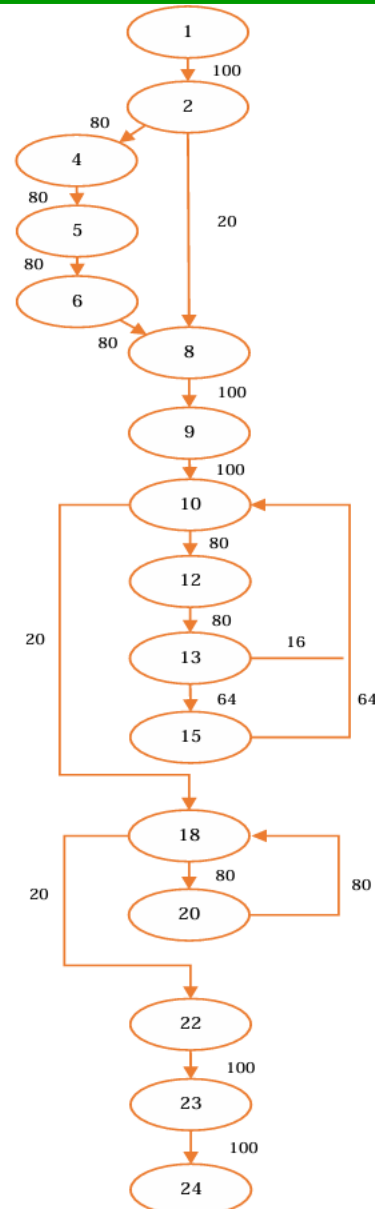
- All nested restrictions are taken into account, for example, for two nested conditions, the weight of operations will be equal to $80 * 80 = 64$ percent

Test data generation with GA

- We introduce the following notation:
 - X – data sets; m – population size, i.e. the number of different variants of input data values; $r(i)$ – the weight of a single operation i ; $F(X)$ – the value of the fitness function for each data set depending on the calculated weights
- The problem is to select the maximum of the objective function: $F(X) = \sum_{i=0}^m r(i) \rightarrow \max$

Test data generation with GA

```
0. public static void Main(int m, int i, int n) {  
1.   intsum = 0, avg = 0;  
2.   if (m < i)  
3.   {  
4.     use_m = m;  
5.   }  
6.   int[] a = new int[m];  
7.   Console.WriteLine("Enter the Array Elements ");  
8.   for (j= 0; j< m; j++)  
9.   {  
10.    a[j] = i;  
11.    if (a[j] > n)  
12.    {  
13.      a[j] = n  
14.    }  
15.  }  
16.  for (k= 0; k < m; k++)  
17.  {  
18.    sum += a[k];  
19.  }  
20.  avg = sum / m;  
21.  Console.WriteLine(" Average is {0}", avg);  
22.  Console.ReadLine();  
23. }
```



Test data generation with GA

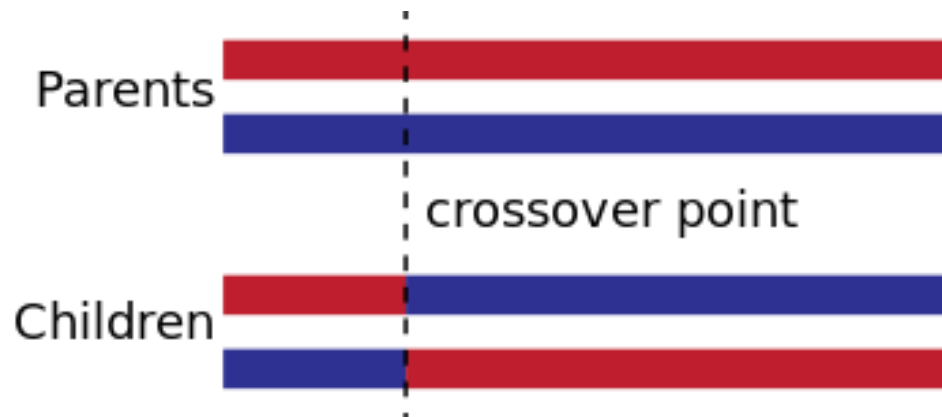
- At the initialization stage we will generate the following data sets – (10,5,12); (3,4,10); (25,30,11); (5,3,17).

Table 1. Initial dataset options.			
N _o	Sets X	F(X)	Rank
1	(10,5,12)	896	3
2	(3,4,10)	1196	2
3	(25,30,11)	1308	1
4	(5,3,17)	896	3

- Table I presents these sets, the calculated target value of the fitness function and the rank corresponding to the best set
- From the table we can see that the better options for the selection are 2nd and 3rd options. In order to obtain additional two new variants, their values will be mixed with a certain probability of mutation

Test data generation with GA

- One-point crossover
 - A point on both parents' chromosomes is picked, and designated a 'crossover point'. Genes to the right of that point are swapped between the two parent chromosomes. This results in two offspring, each carrying some genetic information from both parents.



Test data generation with GA

- In the crossover stage the division of data sets occur after the first and second positions. For example, when crossing (R1, R2, R3) and (G1, G2, G3), the variables obtained are - (R1, R2, G3), (R1, G2, G3), (G1, G2, R3) and (G1, R2, R3)

Table 2. New datasets obtained by crossing.

№	X	Y	New set	Mutation
1	(3,4,10)	(25,30,11)	(3,4,11)	(3,4,13)
2	(3,4,10)	(25,30,11)	(3,30,11)	(3,30,11)
3	(3,4,10)	(25,30,11)	(25,4,10)	(25,4,10)
4	(3,4,10)	(25,30,11)	(25,30,10)	(25,30,10)

- Depending on the settings of the genetic algorithm, the parental chromosomes can be excluded from consideration

Test data generation with GA

- Mutation will occur with a probability of 0.1 for the chance of changing the value from 1 to the specified value in both directions. Under these conditions, the maximum possible value added during a mutation is 5

Table 2. New datasets obtained by crossing.

№	X	Y	New set	Mutation
1	(3,4,10)	(25,30,11)	(3,4,11)	(3,4,13)
2	(3,4,10)	(25,30,11)	(3,30,11)	(3,30,11)
3	(3,4,10)	(25,30,11)	(25,4,10)	(25,4,10)
4	(3,4,10)	(25,30,11)	(25,30,10)	(25,30,10)

Test data generation with GA

- As a result, two more variants will be added to the additional two parent sets - (3,4,13) and (25,30,10)

Table 3. The first generation of test data.

№	Sets X	F(X)	Rank	Generation
1	(3,4,10)	1196	2	0
2	(25,30,11)	1308	1	0
3	(3,4,13)	1196	2	1
4	(3,30,11)	1308	1	1
5	(25,4,10)	896	3	1
6	(25,30,10)	1308	1	1

- If the two variants have the same rank, priority will be given to the option from the **newer** generation
- The first set was obtained from the first generation, so it will be excluded and there will remain only two options - (3,30,11) and (25,30,10)

Hill Climbing

- A local search optimization algorithm used in Artificial Intelligence (AI)
- Starts with **an initial solution** and iteratively makes small changes to improve it
- Relies on **a heuristic function** to evaluate the quality of the solution
- Continues until a **local maximum** is reached, where no further improvement is possible

Hill Climbing

Variations:

- Steepest Ascent: Evaluates all possible moves from the current solution and selects the best improvement
- First-Choice: Randomly selects a move and accepts it if it leads to an improvement
- Simulated Annealing: Accepts **worse moves** occasionally to avoid local maxima

Hill Climbing Algorithm

Advantages:

- Simple and intuitive, easy to understand and implement
- Efficient in finding local optima quickly
- Can be modified to include additional heuristics or constraints

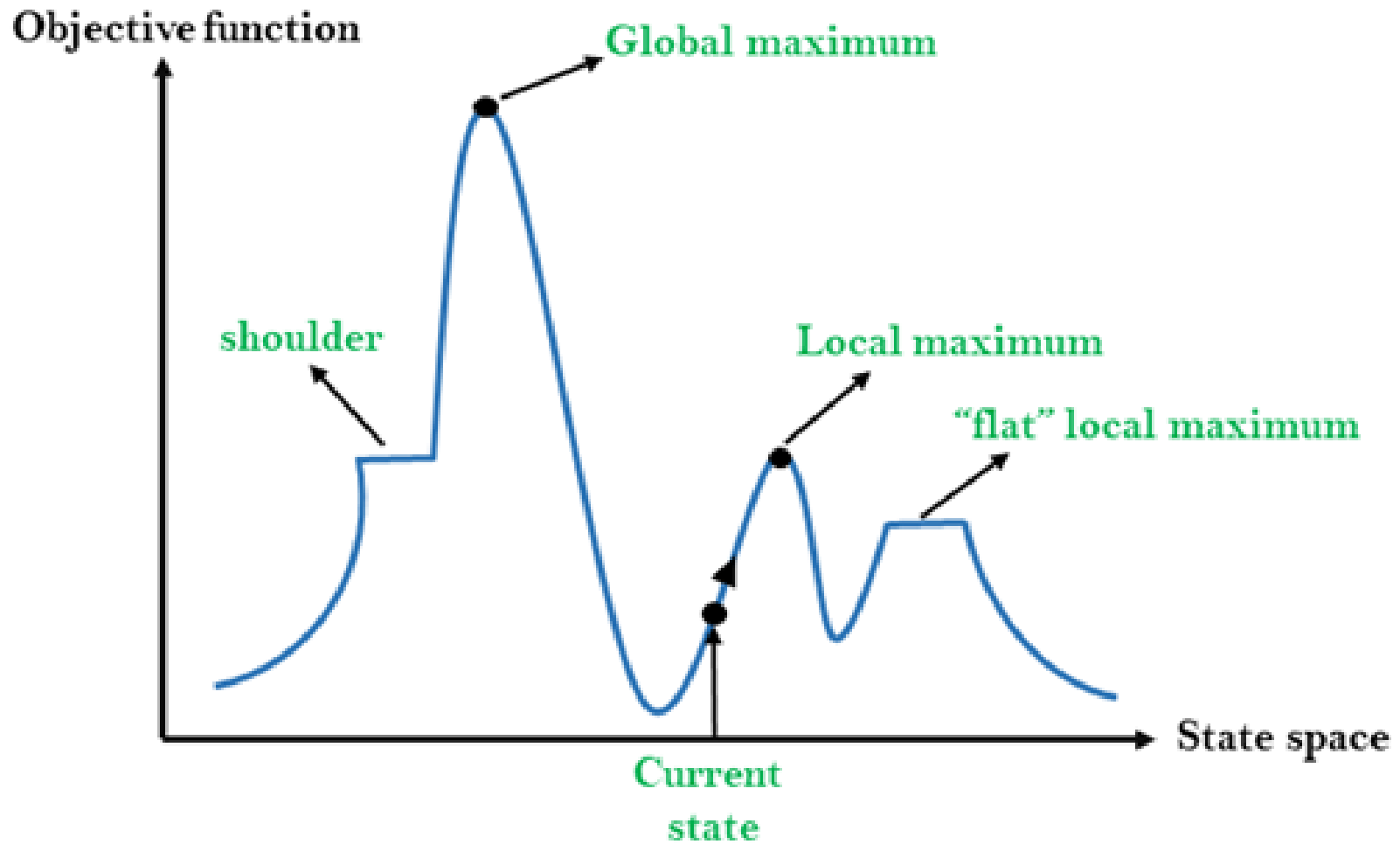
Disadvantages:

- May get stuck in local optima, missing the global optimum
- Sensitive to the choice of initial solution
- Does not explore the search space thoroughly, which can limit its effectiveness

State Space diagram

- The **state-space diagram** is a graphical representation of the set of states our search algorithm can reach vs the value of our objective function (the function which we wish to maximize)
- X-axis: denotes the state space ie states or configuration our algorithm may reach
- Y-axis: denotes the values of objective function corresponding to a particular state
- The best solution will be a state space where the objective function has a maximum value (global maximum)

State Space diagram



Simulated Annealing

Simulated Annealing Algorithm:

- A probabilistic optimization algorithm inspired by the annealing process in metallurgy
- It involves heating and cooling a material to increase the size of its crystals and reduce defects

Mechanism:

- Starts with a high “temperature” to allow exploration of the solution space
- Gradually lowers the temperature to reduce the probability of accepting worse solutions
- Aims to find a global optimum by escaping local optima

Simulated Annealing

Advantages:

- Can escape local optima, increasing the chance of finding a global optimum
- Flexible and adaptable to various types of optimization problems

Disadvantages:

- Requires careful tuning of parameters like initial temperature and cooling schedule
- Performance can be unpredictable and may require multiple runs to find a good solution

Pynguin

- Pynguin is an **automated** unit test generation framework for Python
- The easiest way to obtain Pynguin is by running this command:

```
pip install pynguin
```

- We invoke Pynguin to let it generate test cases
 - We use \ and the line breaks for better readability here, you can just omit them and type everything in one line:

```
pynguin \  
--project-path ./source \  
--output-path ./pynguin-results \  
--module-name example
```

Pynguin

- To have some more verbose output we add the `-v` parameter:

```
pynguin \  
--project-path ./source \  
--output-path ./pynguin-results \  
--module-name example \  
-v
```

- When you run it for the first time, you'll get this message:

```
Environment variable 'PYNGUIN_DANGER_AWARE' not set.  
Aborting to avoid harming your system.  
Please refer to the documentation  
(https://pynguin.readthedocs.io/en/latest/user/quickstart.html)  
to see why this happens and what you must do to prevent it.
```

Pynguin

- This is because:
 - Pynguin actually executes the code of the module under test
 - That means, if the code you want to generate tests for does something bad, for example wipes your disk, **there is nothing that prevents it from doing so!**
 - Pynguin will immediately abort unless the environment variable **PYNGUIN_DANGER_AWARE** is set
- Set the environment variable:

```
Environment variable 'PYNGUIN_DANGER_AWARE' not set.  
Aborting to avoid harming your system.  
Please refer to the documentation  
(https://pynguin.readthedocs.io/en/latest/user/quickstart.html)  
to see why this happens and what you must do to prevent it.  
erfan@Erfan /m/d/nf [255]> export PYNGUIN_DANGER_AWARE=x
```


Pynguin

- Now we invoke Pynguin again to generate test cases for queue class:

```
1  import array
2
3
4  class Queue:
5      def __init__(self, size_max: int) -> None:
6          assert size_max > 0
7          self.max = size_max
8          self.head = 0
9          self.tail = 0
10         self.size = 0
11         self.data = array.array("i", range(size_max))
12
13     def empty(self) -> bool:
14         return self.size != 0
15
16     def full(self) -> bool:
17         return self.size == self.max
18
```

```
19
20     def enqueue(self, x: int) -> bool:
21         if self.size == self.max:
22             return False
23         self.data[self.tail] = x
24         self.size += 1
25         self.tail += 1
26         if self.tail == self.max:
27             self.tail = 0
28         return True
29
30     def dequeue(self) -> int | None:
31         if self.size == 0:
32             return None
33         x = self.data[self.head]
34         self.size -= 1
35         self.head += 1
36         if self.head == self.max:
37             self.head = 0
38         return x
39
```

Pynguin

- Pynguin report:

```
INFO      Start Pynguin Test Generation...
INFO      Collecting static constants from module under test
INFO      No constants found
INFO      Setting up runtime collection of constants
INFO      Analyzed project to create test cluster
INFO      Modules:      2
INFO      Functions:    0
INFO      Classes:      13
INFO      Using seed 1716993507032279516
INFO      Using strategy: Algorithm.DYNAMOSA
INFO      Instantiated 14 fitness functions
INFO      Using CoverageArchive
INFO      Using selection function: Selection.TOURNAMENT_SELECTION
INFO      No stopping condition configured!
INFO      Using fallback timeout of 600 seconds
INFO      Using crossover function: SinglePointRelativeCrossOver
INFO      Using ranking function: RankBasedPreferenceSorting
```

Project

Test generation algorithm

Pynguin

- Pynguin report:

```
INFO      Using strategy: Algorithm.DYNAMOSA
INFO      Instantiated 14 fitness functions
INFO      Using CoverageArchive
INFO      Using selection function: Selection.TOURNAMENT_SELECTION
INFO      No stopping condition configured! → Stop condition
INFO      Using fallback timeout of 600 seconds
INFO      Using crossover function: SinglePointRelativeCrossOver
INFO      Using ranking function: RankBasedPreferenceSorting
INFO      Start generating test cases
INFO      Initial Population, Coverage: 0.857143
INFO      Iteration:      1, Coverage: 0.857143
INFO      Iteration:      2, Coverage: 1.000000
INFO      Algorithm stopped before using all resources.
INFO      Stop generating test cases
INFO      Start generating assertions
INFO      Setup mutation controller
INFO      Build AST for queue_example
INFO      Mutate module queue_example
:28] INFO      Generated 31 mutants
INFO      Running tests on mutant 1/31
INFO      Running tests on mutant 2/31
INFO      Running tests on mutant 3/31
```

Number of iterations and the achieved branch coverage

Pynguin

- With each iteration the algorithm tries to increase branch coverage

```
INFO      Start generating test cases
INFO      Iteration:      1, Coverage: 0.142857
INFO      Iteration:      2, Coverage: 0.142857
INFO      Iteration:      3, Coverage: 0.142857
INFO      Iteration:      4, Coverage: 0.142857
INFO      Iteration:      5, Coverage: 0.142857
INFO      Iteration:      6, Coverage: 0.214286
INFO      Iteration:      7, Coverage: 0.357143
```

Pynguin

- Output (test) file:

```
1  # Test cases automatically generated by Pynguin (https://www.pynguin.eu).
2  # Please check them before you use them.
3  import pytest
4  import queue_example as module_0
5
6
7  def test_case_0():
8      bool_0 = True
9      queue_0 = module_0.Queue(bool_0)
10     assert (
11         f"{type(queue_0).__module__}.{type(queue_0).__qualname__}"
12         == "queue_example.Queue"
13     )
14     assert queue_0.max is True
15     assert queue_0.head == 0
16     assert queue_0.tail == 0
17     assert queue_0.size == 0
18     assert (
19         f"{type(queue_0.data).__module__}.{type(queue_0.data).__qualname__}"
20         == "array.array"
21     )
22     assert len(queue_0.data) == 1
23     var_0 = queue_0.dequeue()
24
25
26  def test_case_1():
27      bool_0 = False
28      with pytest.raises(AssertionError):
29          module_0.Queue(bool_0)
30
```


Pynguin

- Output (test) file:

```
32  ✓ def test_case_2():
33      bool_0 = True
34      queue_0 = module_0.Queue(bool_0)
35  ✓  assert (
36      |      f"{type(queue_0).__module__}.{type(queue_0).__qualname__}"
37      |      == "queue_example.Queue"
38      |  )
39      assert queue_0.max is True
40      assert queue_0.head == 0
41      assert queue_0.tail == 0
42      assert queue_0.size == 0
43  ✓  assert (
44      |      f"{type(queue_0.data).__module__}.{type(queue_0.data).__qualname__}"
45      |      == "array.array"
46      |  )
47      assert len(queue_0.data) == 1
48      bool_1 = queue_0.enqueue(bool_0)
49      assert bool_1 is True
50      assert queue_0.size == 1
51
52
53      @pytest.mark.xfail(strict=True)
54  ✓ def test_case_3():
55      none_type_0 = None
56      module_0.Queue(none_type_0)
```

Pynguin

- As you can see Pynguin generated test suite for the source code
- Pynguin has a lot of parameters and configurations:
- Different algorithms supported by Pynguin:
 - *DYNAMOSA* = Dynamic Many-Objective Sorting Algorithm
 - *MIO* = Many Independent Objective
 - *MOSA* = Many-Objective Sorting Algorithm
 - *RANDOM* = A feedback-direct random test generation approach similar to the algorithm proposed by Randoop
 - *RANDOM_TEST_CASE_SEARCH* = Performs random search on test cases

Pynguin

- *RANDOM_TEST_SUITE_SEARCH* = Performs random search on test suites
- *WHOLE_SUITE* = A whole-suite test generation approach similar to the one proposed by EvoSuite
- **DYNAMOSA** is the **default** test generation algorithm of Pynguin
- Different approaches for assertion generation supported by Pynguin:
 - *CHECKED_MINIMIZING* = All assertions that do not increase the checked coverage are removed
 - *MUTATION_ANALYSIS* = Use the mutation analysis approach for assertion generation

Pynguin

- *NONE* = Do not create any assertions
- *SIMPLE* = Use the simple approach for primitive and none assertion generation
- You can even specify the stop condition:
 - *stopping* = Stopping configuration
- The different available coverage metrics available for optimization:
 - *BRANCH* = Calculate how many of the possible branches in the code were executed
 - *CHECKED* = Calculate how many of the possible lines in the code are checked by an assertion
 - *LINE* = Calculate how many of the possible lines in the code were executed

Pynguin

- We set maximum coverage = 70
- Algorithm = MIO

```
pynguin \  
  --project-path . \  
  --output-path ./fail \  
  --module-name queue_example \  
  -v \  
  --algorithm MIO \  
  --maximum_coverage=70 \  
  
```

```
INFO    Iteration:    151, Coverage: 0.642857  
INFO    Iteration:    152, Coverage: 0.642857  
INFO    Iteration:    153, Coverage: 0.642857  
INFO    Iteration:    154, Coverage: 0.642857  
INFO    Iteration:    155, Coverage: 0.642857  
INFO    Iteration:    156, Coverage: 0.642857  
INFO    Iteration:    157, Coverage: 0.642857  
INFO    Iteration:    158, Coverage: 0.642857  
INFO    Iteration:    159, Coverage: 0.642857  
INFO    Iteration:    160, Coverage: 0.642857  
INFO    Iteration:    161, Coverage: 0.642857  
INFO    Iteration:    162, Coverage: 0.642857  
INFO    Iteration:    163, Coverage: 0.642857  
INFO    Iteration:    164, Coverage: 0.642857  
INFO    Iteration:    165, Coverage: 0.642857  
INFO    Iteration:    166, Coverage: 0.642857  
INFO    Iteration:    167, Coverage: 0.642857  
INFO    Iteration:    168, Coverage: 0.642857  
INFO    Iteration:    169, Coverage: 0.642857  
INFO    Iteration:    170, Coverage: 0.642857  
INFO    Iteration:    171, Coverage: 0.642857  
INFO    Iteration:    172, Coverage: 0.642857  
INFO    Iteration:    173, Coverage: 0.642857  
INFO    Iteration:    174, Coverage: 0.642857  
INFO    Iteration:    175, Coverage: 0.642857  
INFO    Iteration:    176, Coverage: 0.642857  
INFO    Iteration:    177, Coverage: 0.642857  
INFO    Iteration:    178, Coverage: 0.642857  
INFO    Iteration:    179, Coverage: 0.785714  
INFO    Stopping condition reached  
INFO    Achieved coverage: 1.114286%
```

Pynguin

- Without maximum coverage:

```
pynguin \
  --project-path . \
  --output-path ./fail \
  --module-name queue_example \
  -v \
  --algorithm MIO \
```

```
INFO      Iteration:      185, Coverage: 0.928571
INFO      Iteration:      186, Coverage: 0.928571
INFO      Iteration:      187, Coverage: 0.928571
INFO      Iteration:      188, Coverage: 0.928571
INFO      Iteration:      189, Coverage: 0.928571
INFO      Iteration:      190, Coverage: 0.928571
INFO      Iteration:      191, Coverage: 0.928571
INFO      Iteration:      192, Coverage: 0.928571
INFO      Iteration:      193, Coverage: 0.928571
INFO      Iteration:      194, Coverage: 0.928571
INFO      Iteration:      195, Coverage: 0.928571
INFO      Iteration:      196, Coverage: 0.928571
INFO      Iteration:      197, Coverage: 0.928571
INFO      Iteration:      198, Coverage: 0.928571
INFO      Iteration:      199, Coverage: 0.928571
INFO      Iteration:      200, Coverage: 0.928571
INFO      Iteration:      201, Coverage: 0.928571
INFO      Iteration:      202, Coverage: 0.928571
INFO      Iteration:      203, Coverage: 0.928571
INFO      Iteration:      204, Coverage: 0.928571
INFO      Iteration:      205, Coverage: 0.928571
INFO      Iteration:      206, Coverage: 0.928571
INFO      Iteration:      207, Coverage: 0.928571
INFO      Iteration:      208, Coverage: 0.928571
INFO      Iteration:      209, Coverage: 0.928571
INFO      Iteration:      210, Coverage: 0.928571
INFO      Iteration:      211, Coverage: 0.928571
INFO      Iteration:      212, Coverage: 0.928571
INFO      Iteration:      213, Coverage: 1.000000
INFO      Algorithm stopped before using all resources.
INFO      Stop generating test cases
```

Mutation Testing

- After reaching the set coverage metric, Pynguin tries to eliminate irrelevant test cases using **Mutation Analysis (Mutation Testing)**
- Mutation testing is done by selecting a set of mutation operators and then applying them to the source program one at a time
- The result of applying one mutation operator to the program is called a mutant
- If the test suite is able to detect the change (one of the tests fails), then the mutant is said to be killed

Mutation Testing

- For example, consider the following C++ code fragment
- We replace `&&` with `||` and produce the following mutant:

```
if (a && b) {  
    c = 1;  
} else {  
    c = 0;  
}
```



```
if (a || b) {  
    c = 1;  
} else {  
    c = 0;  
}
```

1. A test must reach the mutated statement
 2. Test input data should infect the program state by causing different program states for the mutant and the original program (a=1, b=0)
 3. The incorrect program state (the value of 'c') must propagate to the program's output
- Only if there is a mutant that violates an assertion, the assertion is seen as relevant

Pynguin

- Pynguin exports the generated test cases in the style of the PyTest framework
- We can use PyTest to run generated test file

```
erfan@Erfan /m/d/nf [4]> code .  
erfan@Erfan /m/d/nf> cd MIO/  
erfan@Erfan /m/d/n/MIO> pytest
```

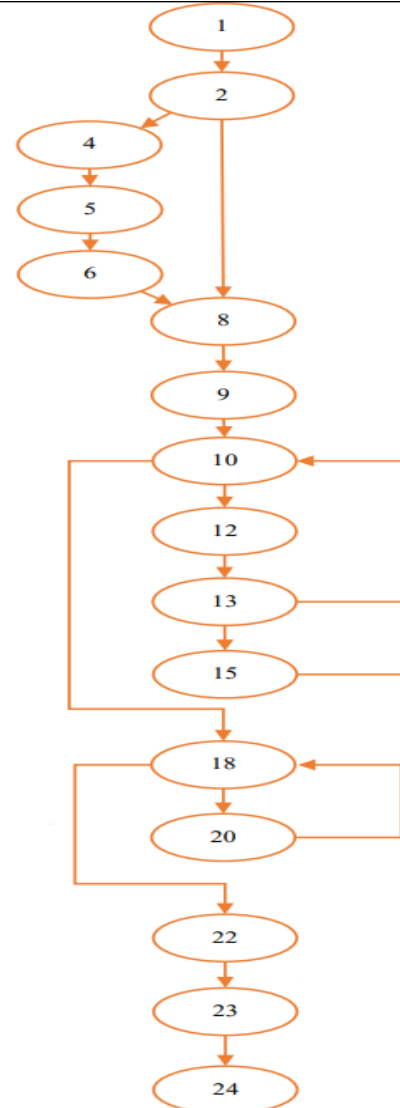
- PyTest report:

```
erfan@Erfan /m/d/n/MIO> pytest  
===== test session starts =====  
platform linux -- Python 3.10.0, pytest-8.2.1, pluggy-1.5.0  
rootdir: /mnt/d/nf/MIO  
plugins: anyio-4.2.0  
collected 11 items  
  
test_queue_example.py .....x..x..  
  
===== 9 passed, 2 xfailed in 0.06s =====
```


Questions

1. Assign weight to the following CFG using genetic algorithm. [Click for Answer](#)

```
0. public static void Main(int m, int i, int n) {  
1.   intsum = 0, avg = 0;  
2.   if (m < i)  
3.   {  
4.     use_m = m;  
5.   }  
6.   int[] a = new int[m];  
7.   Console.WriteLine("Enter the Array Elements ");  
8.   for (j= 0; j< m; j++)  
9.   {  
10.    a[j] = i;  
11.    if (a[j] > n)  
12.    {  
13.      a[j] = n  
14.    }  
15.  }  
16.  for (k= 0; k < m; k++)  
17.  {  
18.    sum += a[k];  
19.  }  
20.  avg = sum / m;  
21.  Console.WriteLine("Average is {0}", avg);  
22.  Console.ReadLine();  
23. }
```



Questions

2. Using genetic algorithm find a dataset for optimal branch coverage. Do one crossover and ignore mutations. [Click for Answer](#)
3. What are the three conditions that determine whether a test is valid or not? [Click for Answer](#)

References

- [1] P. McMinn, “Search-Based Software Testing: Past, Present and Future,” *International Conference on Software Testing, Verification and Validation Workshops*, Mar. 2011, doi: <https://doi.org/10.1109/icstw.2011.100>.
- [2] M. Khari and P. Kumar, “An extensive evaluation of search-based software testing: a review,” *Soft Computing*, vol. 23, no. 6, pp. 1933–1946, Nov. 2017, doi: <https://doi.org/10.1007/s00500-017-2906-y>.
- [3] Konstantin Serdyukov and T. Avdeenko, “Using genetic algorithm for generating optimal data sets to automatic testing the program code,” Jan. 2019, doi: <https://doi.org/10.18287/1613-0073-2019-2416-173-182>.
- [4] “Introduction to Hill Climbing | Artificial Intelligence,” GeeksforGeeks, Dec. 12, 2017. <https://www.geeksforgeeks.org/introduction-hill-climbing-artificial-intelligence/>
- [5] “Pynguin—PYthon General Unit test geNerator — pynguin 0.36.0.dev documentation,” pynguin.readthedocs.io. <https://pynguin.readthedocs.io/en/latest/> (accessed Jun. 02, 2024).