Random Testing

Shin Yoo

Random Testing

- One of the most important task in testing: sampling test input from the enormous input space
- Doing things randomly can be good in testing!
    - Developers often hold onto functional biases about the code they wrote: they see what should work, and not what should not work
    - Random ignores this bias

Some Definitions

- SUT: Software Under Test
- S: set of all possible test inputs for SUT
- |S|: cardinality of S
- F: subset of S - a set of all failing test inputs (not known in advance, of course)
- Failure Rate t = |F|/|S|: the probability that a random test input will fail, if sampled uniformly

Example

- Failure Rate t ≈ 0.5
- Oracle
  - assertEqual(abs(-5), 5)
  - assertEqual(abs(2), 2)

```
int abs(int x)
{
   if(x>0) return x;
   else return x; // should be -x
}
```

Oracle In Action

- Our assertion can be rewritten in a generic form of: `assertEqual(SUT(i), e)`, where:
    - i: test input (e.g. `-5` in `abs(-5)`)
    - e: expected output (e.g. `5` for `abs(-5)`)
- How should we choose the input i?
- Random is *an option* at least.

Random Testing Overview

- Choose test input randomly
- Benefits:
    - Cheap and easy to implement
    - Easy to understand
- It is actually used in the industry (really?)
- There are actually many positive sides

How random can we get?

- True randomness is possible but expensive
- In general, computers use pseudo-random number generators, which generates a long sequence of bits approximating the properties of random bits (the sequence is controlled by a small initial value called *seed*)
- The pseudo-random bits will repeat themselves, but only at very long intervals - okay for testing purposes

java.util.Random

```java
public int nextInt()
{
   return next(32);
}

synchronized protected int next(int bits)
{
   seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
   return (int)(seed >>> (48 - bits));
}
```

Linear Congruential Formula

Probability Distribution

- Input generation becomes random sampling in random testing. but from which probability distribution?
- If we have deep understanding of the target domain as well as our objective, we can bias the distribution. For example, if we definitely suspect that larger input values are more fault prone, we can sample from a distribution that will produce larger numbers more frequently.
- Otherwise, the default can be the uniform distribution. Each test input has the same probability of being chosen: 1 / |S|

Challenges in Random Testing

- Numbers are easy to randomly sample: we just need to decide on a probability distribution.
- Randomly sampling more complicated inputs can be harder than we think.
  - Complex structures
  - Memory and time constraints
  - Input length bias

Complex Data Structures

- What if the input data is a tree/graph/arrays…?
- What is a *random tree*?
- One approach: whatever the high-level data structure is, they are all reprinted in bits, so just stick to random bits: 0000100111101011010 instead of a tree
  - But random trees are not random bits - there definitely is *some* structure in a *random* tree because it is still a *tree*
  - Chance of actually generating a valid input this way might be very low
- Ad-hoc generator approach: make random decisions about *growing* data structure - slightly better, but it may be easy to *bias* the decisions

Memory/Time Constraints

- What is the set of all possible trees? Isn't it *infinite*?
- We need to put constraints on memory/time because we cannot deal with infinity
- If the fault detection is also uniform (i.e. all tests have the same probability to detect a fault), then constraints are okay
  - But is it uniform?
  - For example, when testing a function called `void foo (MyGraph graph)`, will larger graphs lead to higher fault detection? It depends on the type of fault, really…

Test Input Length

- Suppose the test input length can vary, and we can afford binary strings of length up to L with our testing budget

- 2^L is more than the sum of all lower lengths (e.g. $2^3 > 2^2 + 2^1$): so we will sample much more strings with length L then shorter ones!

    - It would be unwise to sample uniformly: there will be bias for longer strings

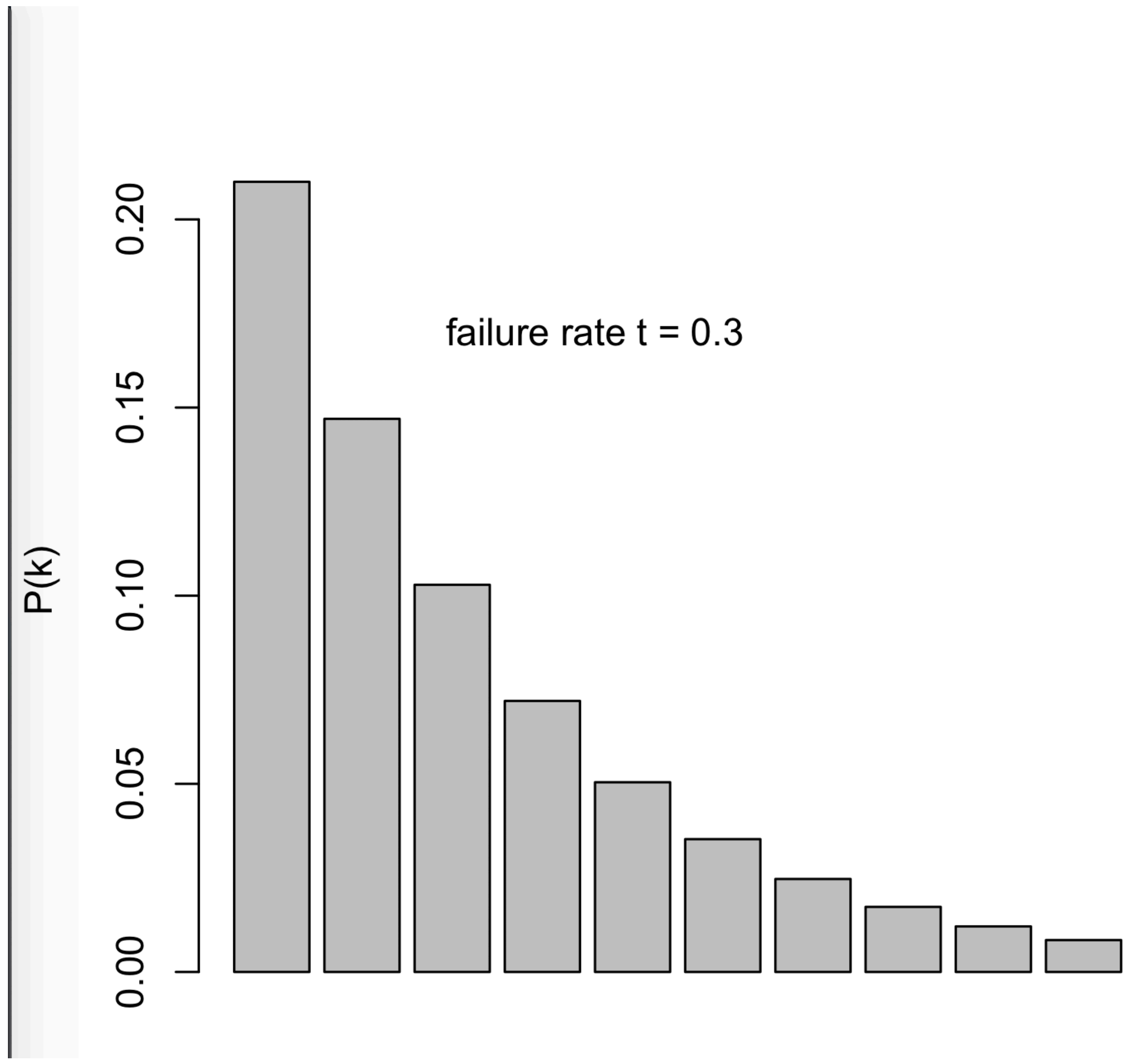
- One possible solution: first choose length randomly, then choose the binary string of that length randomly

Finding Faults with Random Testing

- Properties of a random testing technique depends on the failure rate, t = |F| / |S|
- Given a failure rate t, how many test inputs should we execute until finding the first failure?
- Given k random test inputs, what is the probability of finding at least one failure?

Geometric Distribution of Random Testing

- Probability of revealing a failure at the k-th random input: k-1 inputs should not fail, but the next one does.
- This is geometric distribution (i.e., random testing is a Bernoulli trial, as test either passes or rails)



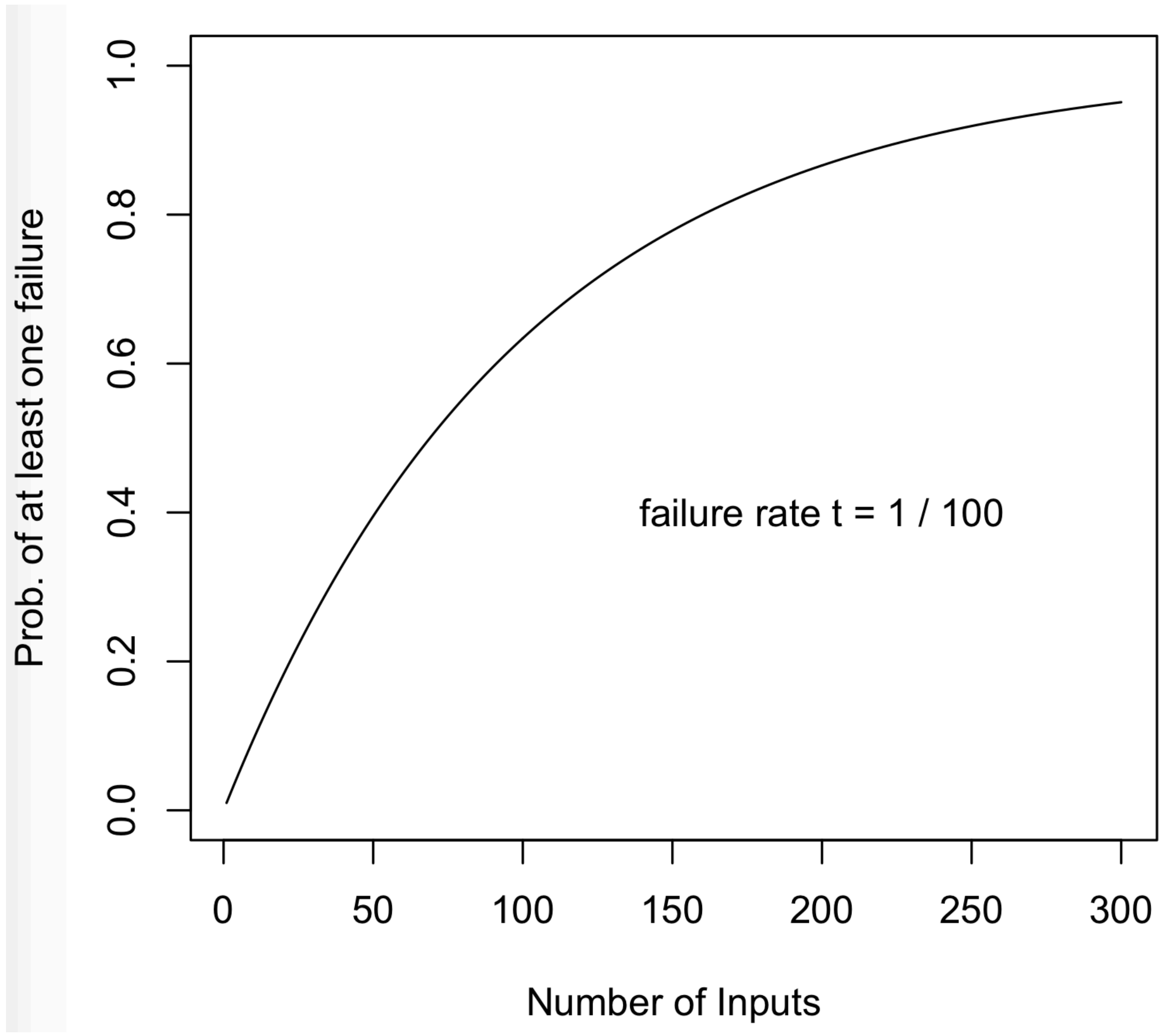
Number of Inputs Until First Failure

- Geometric Distribution:
  - Mean: $1/p$
  - Median: $\lceil \frac{-1}{\log_2(1-t)} \rceil$
  - Variance: $\frac{1-p}{p^2}$

- E.g, when t = 1 / 100
  - Mean: 100
  - Median: ≃69
  - Variance: 9900

Probability of Finding At Least One Failure

- Practically, we may be more interested in the following question: if I execute k random inputs, what is the probability of finding at least one failure?
- This is equal to 1 - the probability of none of the k inputs failing:



“But t is unknown - useless!”

- True, but it can be estimated in various ways
  - Previous projects
  - Literature
  - Types of software
- And this type of analysis is still important to understand the dynamics of random testing

Weakness: No Guidance

- Random testing is particularly weak against the needle in the haystack problem

- Consider the example: we need, on average, over 4 billion test inputs before triggering this fault.

- What if the predicate is `(x == y)`?

```
void foo(int x){
if(x==0){
   /* faulty code here */}
}
```

```
void foo(int x, int y){
if(x==y){
   /* faulty code here */}
}
```

Weakness: Oracle

- Remember `assertEqual(abs(-5), 5)`?
- If we use a random integer instead of -5, how do we know what to expect? Oracle Problem!
  - A single assertion is okay when the input value is manually pre-defined and the tester completely understands the program semantics (i.e. tester writes the expected value)
  - When the input value is randomly sampled, programer cannot write the corresponding expected value
    - Should the programmer write a program to produce the expected val… wait a minute…

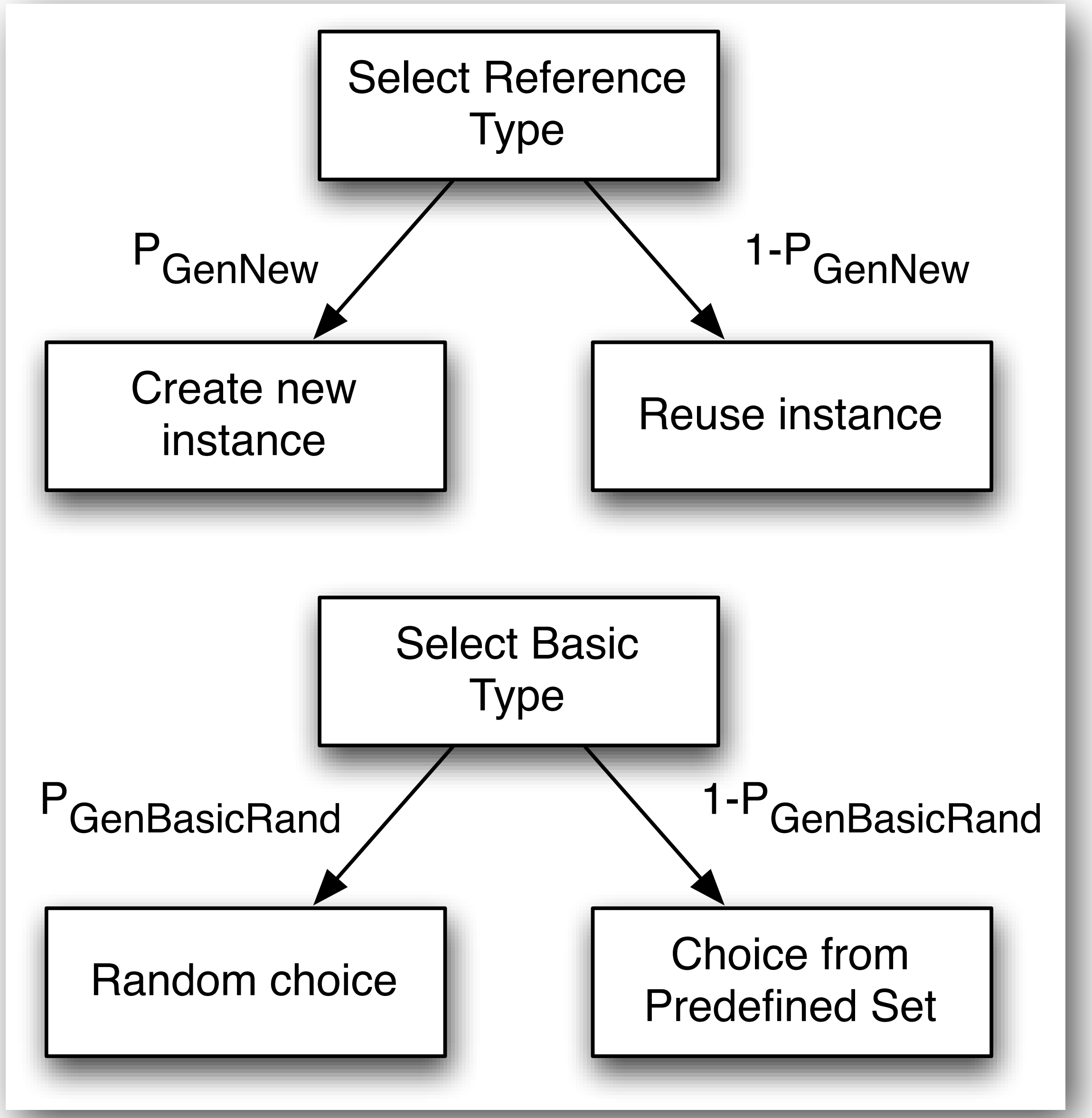
Real World Applications

- Nevertheless, Random Testing is actually used by industry. For example:
  - Randomized Differential Testing as a Prelude to Formal Verification, Groce et al., 2007: random testing for flight control system used in space missions
    - A very complex SUT, for which the formal verification did not work: random testing found multiple faults with automation.

A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In 29th International Conference on Software Engineering (ICSE'07), pages 621–631, May 2007.

Overcoming Weakness

- Introduce bias!
- Instead of random values, use special values (0, 1, Pi,...), extracted values (constants from the code), or previously successful values; use these with predetermined probability parameter
- Helps dealing with needle in the haystack problem
- Ciupa, I., Leitner, A., Oriol, M., Meyer, B., "Experimental Assessment of Random Testing for Object-Oriented Software", Proceedings of ISSTA'07: International Symposium on Software Testing and Analysis 2007, (London, UK), July 2007



Overcoming The Oracle Problem

- The Oracle problem, on the other hand, does not go away so easily: random testing only truly succeeds with a full oracle (i.e. one can answer questions against random test input)

    - Groce et al. had a reference file system alongside the one that they tested: whatever the reference says is the expected behaviour.

    - Ciupa et al. worked with Eiffel under Design by Contract paradigm: each method has pre- and post-condition, violation of which is a fault.

- Sometimes implicit oracle is sufficient, too.

Netflix ChaosMonkey

- Netflix relies on Amazon Web Service cloud infrastructure to stream videos: the servers scale automatically according to the load
- Aim: detect instance failure in Auto Scaling Group (ASG)
- Action: ChaosMonkey just randomly shuts down AWS instances!
- Oracle: the service should not collapse (implicit)
  - Load balancers should detect the instance failure, re-route requests, and additional instances need to be brought in
- http://techblog.netflix.com/2012/07/chaos-monkey-released-into-wild.html

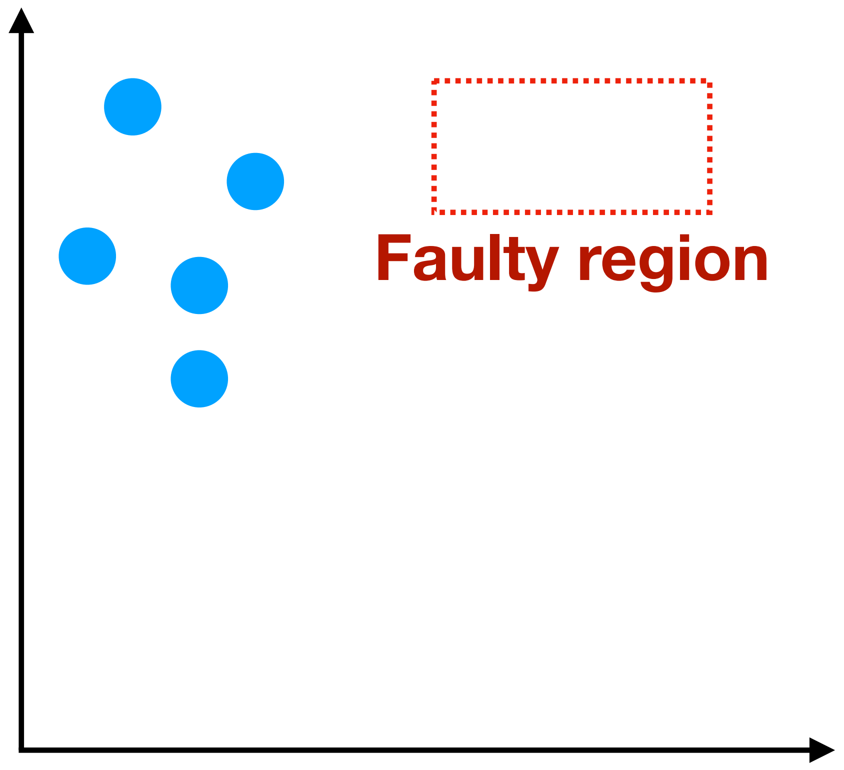
Can we improve it even more?

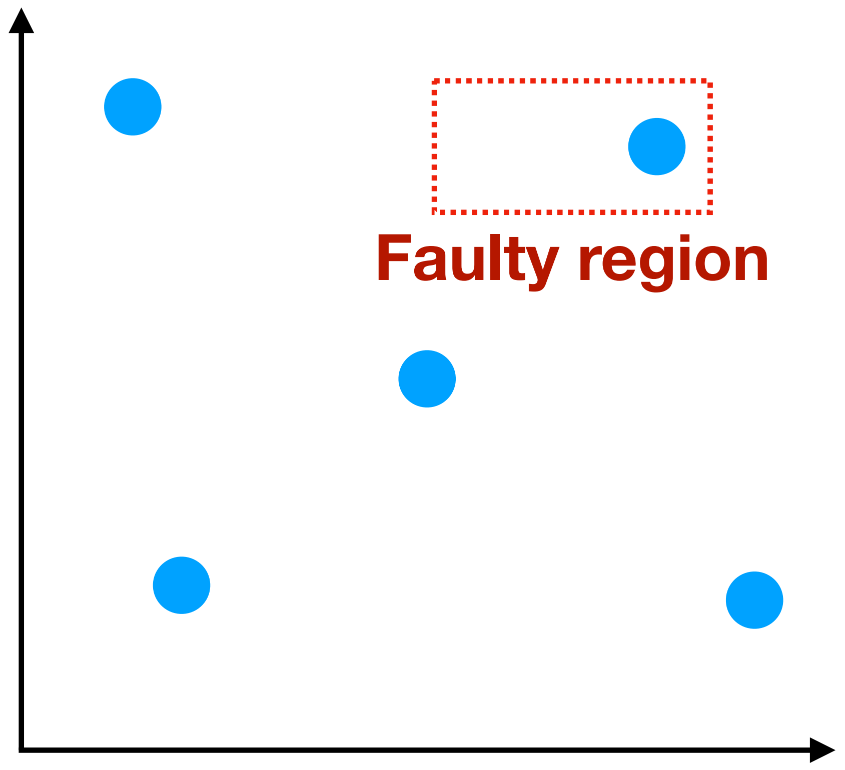
- Given a function `void foo(int n)`, consider two following sets of test inputs:
    - {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
    - {-2345,12,342,-4443,2,3495437,-222223,24, 99343256,-524474}
    - Which is better? Why?
- Without any further assumption, two sets should have identical fault detection capability

Where the faulty things are

- Often, failing test inputs cluster together
  - What if the fault is under the predicate `if(x>=0 && x<=10)`
- Let us call these faulty regions in the input space
- Without knowing where they actually are, what is our best strategy?

Diversity





A more diverse set of inputs will have a higher probability of hitting the faulty region

Distance

- Diversity depends on the distance between test inputs
- If input data is numerical, we can use Euclidean distance
- But how do we measure distance between complex data types?

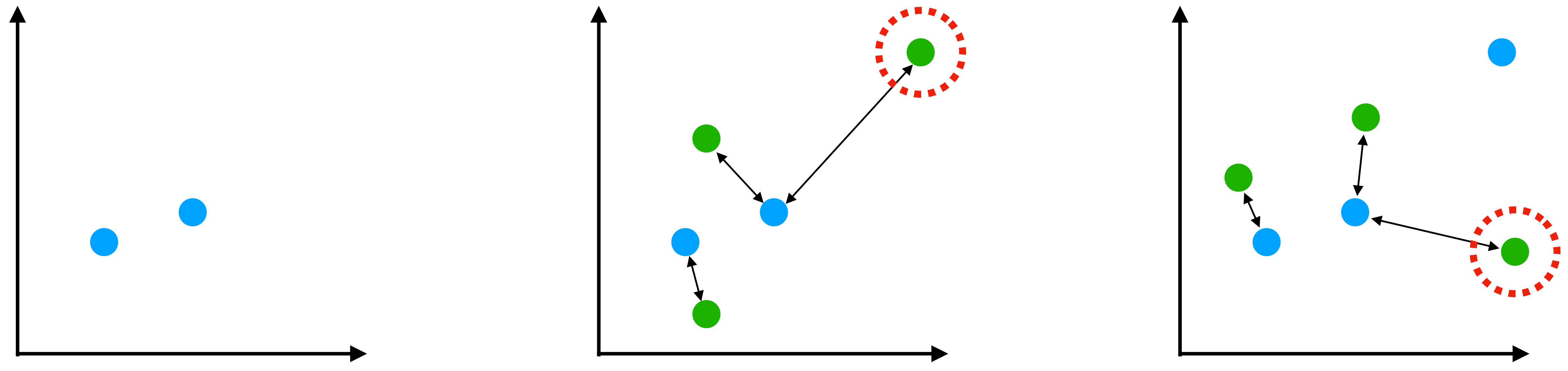
Hamming Distance (Edit Distance)

- Between two strings, Hamming distance is the number of edits that are required to change one string into the other
    - s1 = (A,B,C,A,A), s2 = (A,C,C,A,B) : Hamming distance is 2
- Many other distance metrics are defined between two symbol sequences (DNA sequencing literature is a goldmine)

Adaptive Random Testing

- Still choose test inputs randomly, but only one at a time, into the set G (initially empty)
- Whenever adding a new input:
    - Sample Z inputs randomly
    - Choose z from the Z so that it is the most diverse against (i.e. the farthest away from) the current chosen set G
    - Add z into G

Adaptive Diversity



●	**Existing Input**
●	**New Sample Batch**
◌	**Chosen**

Complexity of ART

- If we use |Z| sample points and get ART test suite of k test cases, how many distance calculations do we need?
  - $0 + |Z| + 2|Z| + 3|Z| + \ldots + (k-1)|Z| = |Z|k(k-1)/2$
  - $O(k^2)$: considering that ART is still random, k may have to be significantly large to detect faults - this can result in very high cost

Pros, Cons, and Questions

- Still very easy to implement
- It may be difficult to choose the right/meaningful distance metric for your input type
- Faulty regions may not apply to all types of faults
- ART is still mostly an academic idea, with debates going on. Here we will look at one argument against ART's universal strength (I've modified the proof slightly).
    - *Adaptive random testing: an illusion of effectiveness?* by Andrea Arcuri and Lionel Briand, ISSTA 2011

Ideal sampling under 1D

- Assume a 1-dimensional domain D, with a single faulty segment with length of $z = \theta |D|$, where $\theta = \frac{z}{|D|}$ is the failure rate. Given the set of input, K, let us consider it in the ascending order (i.e., $x_i \leq x_{i+1}$).

- Let S_1 be the set of all possible SUTs that take a single integer as an input, grouped by the corresponding z (program groups). That is, $s \in S_1$ is a set of programs that share a same faulty region. It follows that $|S_1| = (|D| - z) + 1$ (i.e., ways of placing the faulty region in D).

Ideal sampling under 1D

- Theorem: given any $s_i \in S_1$, for a program in s_i, assuming one faulty region of size z, a sufficient condition to maximise the probability $p_{S_1,K}$ that a set of K test cases detects a failure is when:

$$\min\left(\operatorname{dist}\left(x_{\min}, x_1\right), \operatorname{dist}\left(x_1, x_2\right), \ldots, \operatorname{dist}\left(x_i, x_{i+1}\right), \ldots, \operatorname{dist}\left(x_{|K|-1}, x_{|K|}\right), \operatorname{dist}\left(x_{|K|}, x_{\max}\right)\right) \geq z$$

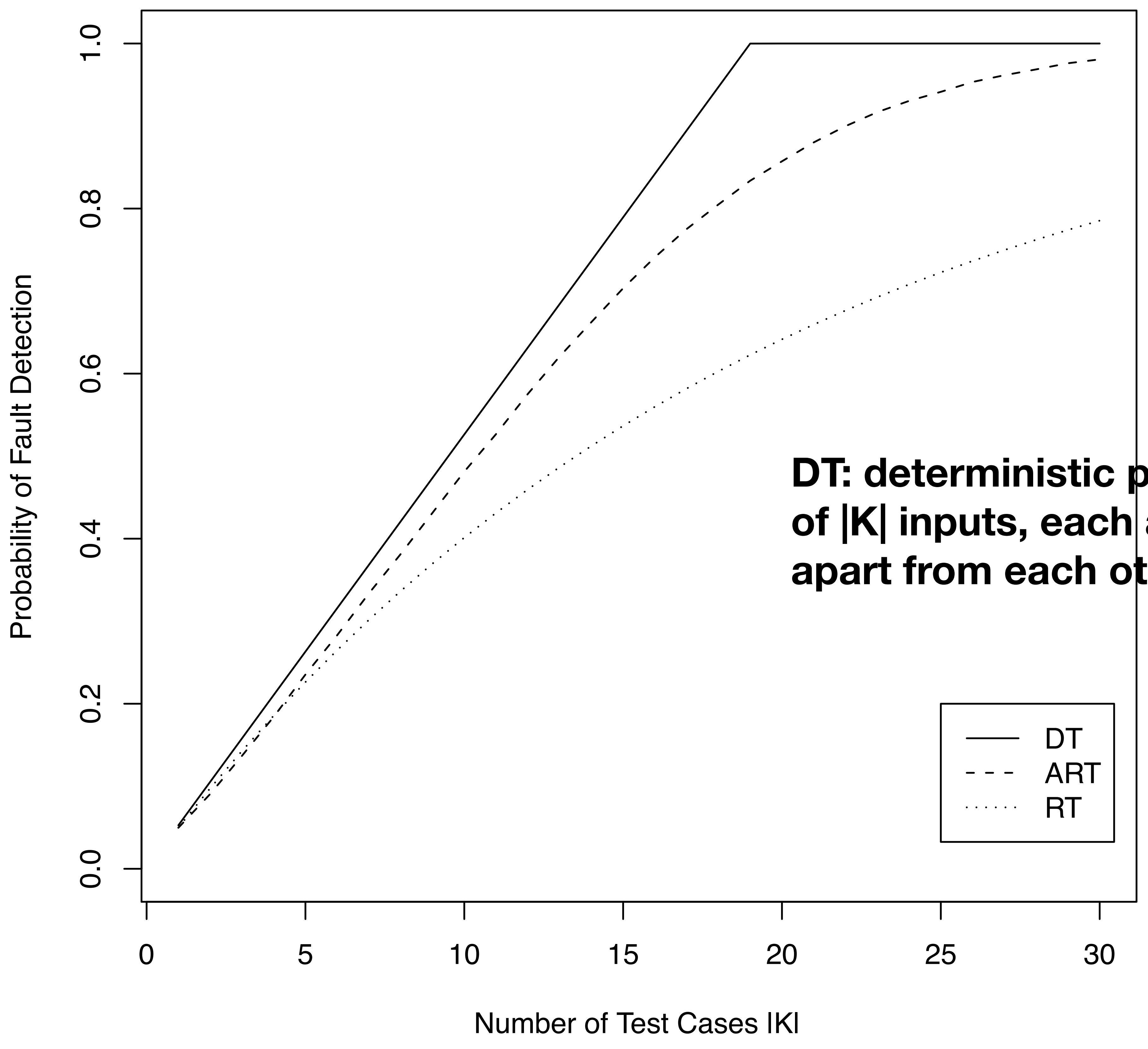
Ideal sampling under 1D

- Proof: Let x_{start} be the starting position of the faulty region F_s, which is $\left[x_{start}, x_{start} + z - 1\right]$. The starting point x_{start} can be position at $(|D| - z) + 1$ places. Consequently, a test data x can reveal failures in at most z program group (fewer if x is near the edges of D). So the upper bound of $p_{S_1,K}$ is $p_{S_1,K} \le \frac{z|K|}{|S_1|}$.

  1. When $\min(\text{dist}(x_{\min}, x_1), \text{dist}(x_{|K|}, x_{\max})) \ge z$, then all inputs in K are at least z away from the edge, therefore each test case in K finds failures in z programs (i.e., z ways of placing F_s from either x_{min} or x_{max} to the end of D).

  2. When $\min(\text{dist}(x_i, x_{i+1})) \ge z$ for all $x_i, x_{i+1} \in K$, each subsequent test input x_{i+1} in the sorted K is outside the faulty region that encapsulates x_i So all test input in K detect faults in different program groups.

- Combining the two cases above, test input in K reveal total $|K|z$ different failures. Consequently, $p_{S_1,K} = \frac{z|K|}{|S_1|}$



DT: deterministic placement of |K| inputs, each at least (|D|-1)/(|K|+1) apart from each other.

Open Questions

- How about non-numeric, 2D or higher space?
- Non-contiguous faulty regions? (although this presents challenges for both DT and ART)

Output Diversity

- An interesting take on testing and diversity: if a test input generates a never-before-seen output, perhaps the test is doing something interesting.

- N. Alshahwan and M. Harman. *Augmenting test suites effectiveness by increasing output diversity*. In Proceedings of the 34th International Conference on Software Engineering, pages 1345–1348, 2012.

  - When adding test cases to coverage-adequate test suites, those that generates unique output outperformed random inputs in terms of fault detection

Property Based Testing

- The PBT idea is originally from the QuickCheck for Haskell, developed in 1999 (http://www.cse.chalmers.se/~rjmh/QuickCheck/)
- It aims to attack the following problems:
  - Random testing can be highly effective, but is weak against structured inputs.
  - Also, automated oracle is essential for effective random testing.
  - Structural coverage itself does not guarantee anything.

Property Based Testing

- PBT can be thought of as the combination of the following:
  - Property based oracles, instead of input-output pair oracles
  - Test input generators that combine low level random generators to build a input generator for complex structured inputs
- Using these two, PBT randomly samples complex, structured inputs, and reports anything that violates the given property.

Hypothesis

- We are going to use Hypothesis in our examples.
- Hypothesis is an easy-to-use PBT framework for Python: https://github.com/HypothesisWorks/hypothesis-python

Property Based Oracles

- Suppose we want to test a Python implementation of the absolute function.
- Traditional, example-based oracle requires test engineers to provide input-output pairs
- For complicated functions, this is tedious and error-prone

```
def abs_function(x):
  if x < 0:
    return -x
  else:
    return x
```

```
def test_important_func():
  assert abs_function(1) == 1
  assert abs_function(0) == 0
  assert abs_function(-1) == 1
```

Property Based Oracles

- Hypothesis allows test engineers to write parameterised unit tests.
- Instead of specifying a concrete input-output pair, use a parameterised input and describe the expected properties of the output using the input symbol.
- For example, for any integer x, abs(x) should be greater than or equal to 0.

```python
def abs_function(x):
  if x < 0:
    return -x
  else:
    return x
```

```python
def test_important_func(x):
  assert abs_function(x) >= 0
```

Input Generator

- Hypothesis uses Python annotation to parameterise the input.
- During the actual test execution, the parameterised input is randomly sampled.
- Anything that violates the assertions will be reported

```python
from hypothesis import given
from hypothesis import strategies as st
import unittest

def abs_function(x):
  if x < 0:
    return -x
  else:
    return x

class TestAbs(unittest.TestCase):
   @given(x = st.integers())
   def test_abs_function(self, x):
      assert abs_function(x) >= 0

if __name__ == '__main__':
    unittest.main()
```

Using Hypothesis

- GitHub Repository: https://github.com/HypothesisWorks/hypothesis
- Installation: use PIP (`pip install hypothesis`)
- Documentation is available from: https://hypothesis.readthedocs.io

PBT: Pros and Cons

- Can be super strong when done right
- Makes you think about what the code should do much harder than the conventional, example-based testing
- For very complicated input type, writing the input generator becomes too difficult: eventually test cases require test cases for them.

Summary

- Random testing is easy to understand/implement, and effective in real world application. However, it requires automated oracle and may take very long for certain problems.
- There are various ways to improve random testing; all of them uses more specific knowledge about the given Software Under Test
- The original slide from Dr. Acuri is available online at: http://www.uio.no/studier/emner/matnat/ifi/INF4290/v10/undervisningsmateriale/INF4290-RandomTesting.pdf