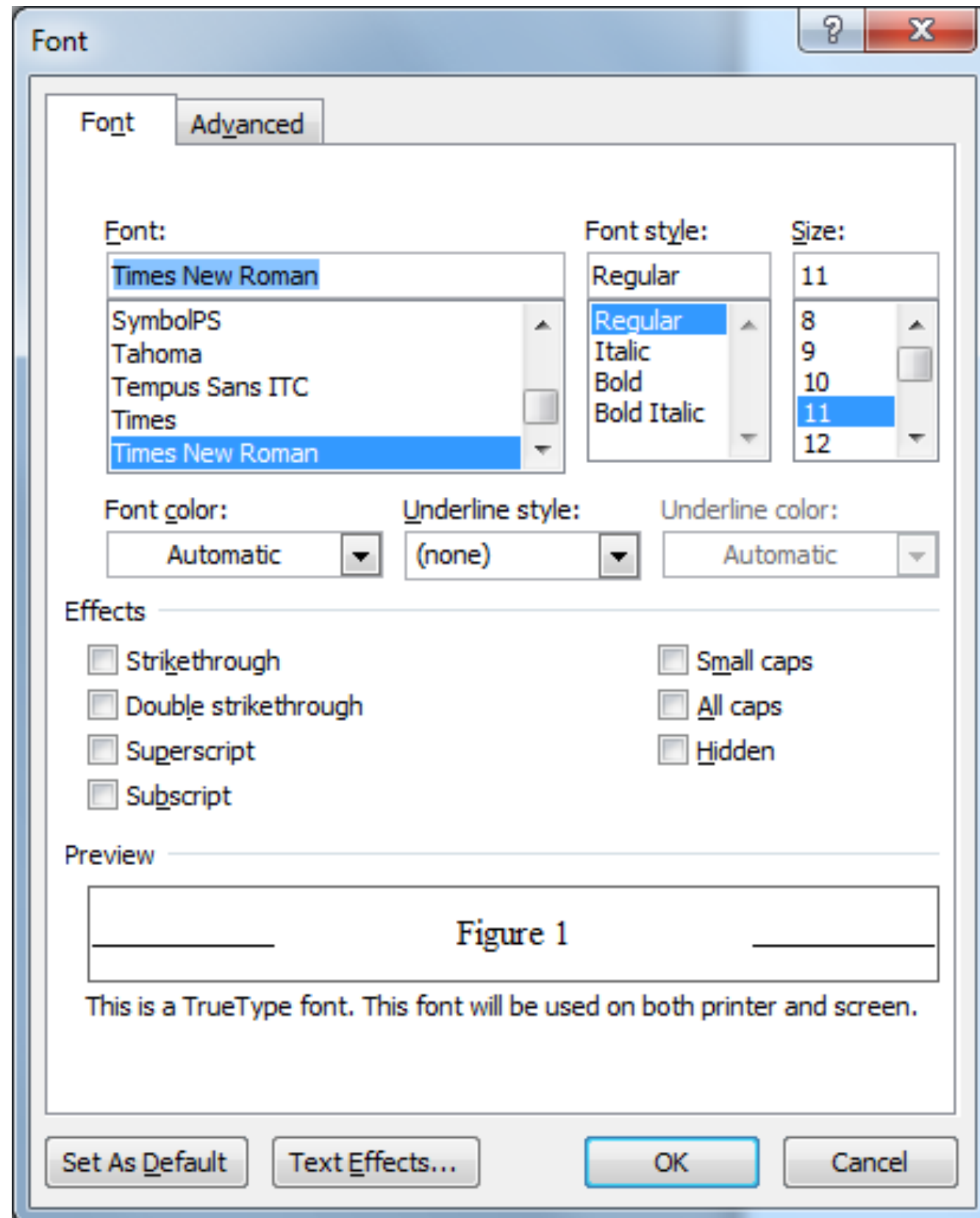


随机测试和 自适应随机测试

Random Testing (RT) and Adaptive Random Testing (ART)



40 Fonts

×

4 Styles

×

30 Sizes

×

10 Colours

×

16 Underlines

×

2⁷ Effects

NAME

ls -- list directory contents

SYNOPSIS

ls [-ABCFGHLOPRSTUW@abcdefghijklmnopqrstuvwxyz1] [file ...]

DESCRIPTION

For each operand that names a file of a type other than directory, **ls** displays its name as well as any requested, associated information. For each operand that names a file of type directory, **ls** displays the names of files contained within that directory, as well as any requested, associated information.

If no operands are given, the contents of the current directory are displayed. If more than one operand is given, non-directory operands are displayed first; directory and non-directory operands are sorted separately and in lexicographical order.

The following options are available:

- @** Display extended attribute keys and sizes in long (-l) output.
- 1** (The numeric digit ``one''.) Force output to be one entry per line. This is the default when output is not to a terminal.
- A** List all entries except for . and ... Always set for the super-user.
- a** Include directory entries whose names begin with a dot (.).
- B** Force printing of non-printable characters (as defined by ctype(3) and current locale settings) in file names as \xxx, where xxx is the numeric value of the character in octal.

隐私设置

- 内容设置...
- 清除浏览数据...

Google Chrome 浏览器可能会使用网络服务改善您的浏览体验。 您可以视情况停用这些服务。 [了解详情](#)

- ☒ 使用网络服务帮助解决导航错误
- ☒ 在地址栏或应用启动器搜索框中输入搜索字词和网址时，借助联想查询服务自动补齐相关内容
- ☒ 预提取资源，以便更快速地加载网页
- ☐ 自动向Google报告可能出现的安全事件详情
- ☒ 启用针对网上诱骗和恶意软件的防护功能
- ☐ 使用网络服务帮助解决拼写错误
- ☐ 将使用情况统计信息和崩溃报告自动发送给 Google
- ☐ 随浏览流量一起发送“请勿跟踪”请求

密码和表单

- ☒ 启用自动填充功能，以便点按一次即可填写网络表单。 [管理自动填充设置](#)
- ☒ 询问是否保存您在网页上输入的密码。 [管理密码](#)

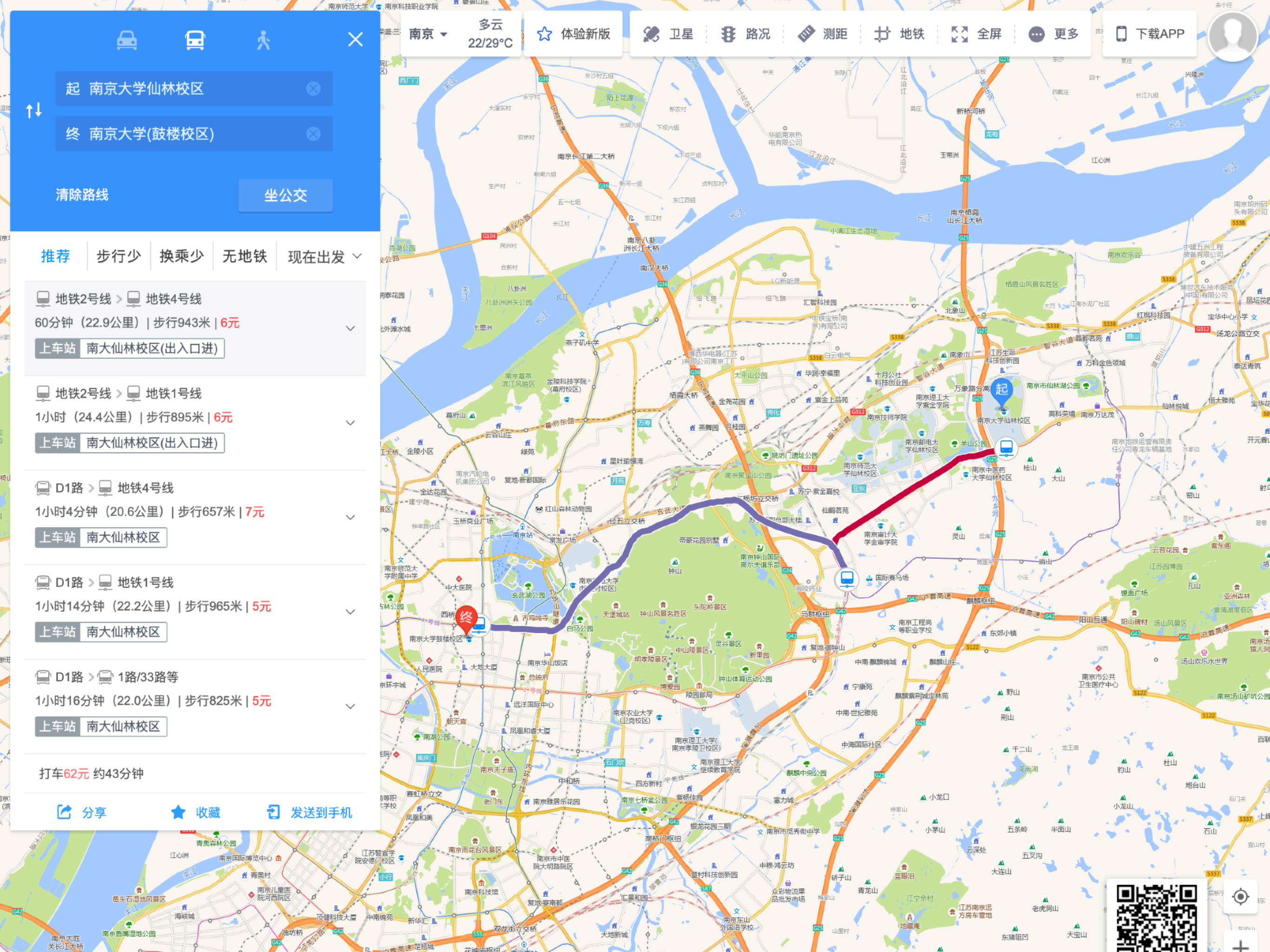
网络内容

- 字号：

中

自定义字体...
- 网页缩放：

100%



南京

多云
22/29°C

体验新版

卫星

路况

测距

地铁

全屏

更多

下载APP



起 南京大学仙林校区

终 南京大学(鼓楼校区)

清除路线

坐公交

推荐

步行少

换乘少

无地铁

现在出发

地铁2号线 > 地铁4号线

60分钟 (22.9公里) | 步行943米 | 6元

上车站 南大仙林校区(出入口进)

地铁2号线 > 地铁1号线

1小时 (24.4公里) | 步行895米 | 6元

上车站 南大仙林校区(出入口进)

D1路 > 地铁4号线

1小时4分钟 (20.6公里) | 步行657米 | 7元

上车站 南大仙林校区

D1路 > 地铁1号线

1小时14分钟 (22.2公里) | 步行965米 | 5元

上车站 南大仙林校区

D1路 > 1路/33路等

1小时16分钟 (22.0公里) | 步行825米 | 5元

上车站 南大仙林校区

打车62元 约43分钟

分享

收藏

发送到手机



随机测试

- 🤔 使用所有可能的输入进行测试是不可行的
- 🤔 程序员在编写代码时往往会有某种功能性偏见
(they see what should work, and not what should not work)
- 💡 随机挑选测试用例 (ignore the bias)

随机测试

看起来很简单，但确实有效 (工业界真实应用)

Randomized Differential Testing as a Prelude to Formal Verification

Alex Groce, Gerard Holzmann, and Rajeev Joshi
Laboratory for Reliable Software *
Jet Propulsion Laboratory
California Institute of Technology
Pasadena, CA 91109. USA

Abstract

Most flight software testing at the Jet Propulsion Laboratory relies on the use of hand-produced test scenarios and executed on systems as similar as possible to actual mission hardware. We report on a flight software development effort incorporating large-scale (biased) randomized testing on commodity desktop hardware. The results show that use of a reference implementation, hardware simulation with fault injection, a testable design, and test minimization enabled a high degree of automation in fault detection and correction.

Our experience will be of particular interest to developers and researchers working in domains where on-time delivery of software is critical (a strong argument for randomized automated testing) but not at the expense of correctness and reliability (a strong argument for model checking, theorem proving, and other heavyweight techniques). The effort spent in randomized testing can prepare the way for generating more complete confidence.

1. Introduction

Every space mission generates a of large amount of data, which must be stored until it can be downlinked to Earth for analysis (or display on NASA TV and websites). In recent missions, JPL has relied on *Flash memory* to store most of this data, as Flash uses little power, has little mass, has a

reliability for space missions. JPL's experience with commercial Flash systems has not established confidence that that current commerical Flash file systems provide the reliability required for space mission use [13].

Flash devices in space operate in a hostile environment: solar flares and higher radiation levels make hardware faults more likely; system reboot is a standard fault protection response (as shown in the MER Flash anomaly [13]), making it essential that a mission file system survive reboots without loss of data integrity.

The Laboratory for Reliable Software has been developing Flash file system software for use in future space missions, with the goal of using the full array of available software verification and validation technologies — randomized testing, runtime monitoring, static analysis, model checking, and theorem proving [3] — to establish confidence in the correctness of this file system.

This paper describes our efforts with randomized differential testing, including our approach to hardware simulation with fault injection, randomized test input generation, design for testability, and automated test case minimization. Our eventual goals encompass more formal, heavyweight approaches to correctness, but past experience (and common sense) indicate that during the early stages of development, when requirements and code are fluid and rich in errors, systematic, disciplined *testing* with a high degree of automation offers a much quicker path to finding many bugs. The infrastructure and specification effort applied to testing will pay off in more rigorous approaches (e.g. model

- 测试 Flight Control System (Space Missions)
- 系统复杂性导致无法应用形式化验证
- 使用随机测试发现故障

A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. International Conference on Software Engineering (ICSE), 621–631, 2007. 7

随机测试

看起来很简单，但确实有效 (工业界真实应用)

Netflix/chaosmonkey

chaosmonkey — Chaos Monkey is a resiliency tool that helps applications tolerate random instance failures.

github.com



- Netflix 依赖亚马逊云服务 (AWS) 基础设施来传输视频，但 AWS 实例可能会以不可预测的方式发生故障
- 应用混沌工程 (Chaos Engineering) 来确保系统的韧性 (resilience)
 - 随机关闭一些服务实例 (via fault injection)
 - 系统的关键服务不应奔溃 (在检测到服务实例故障时，load balancing 应能重新发送请求，并根据需要引入额外的运行实例)

随机测试的缺陷检测能力

失效率 (Failure Rate) 是研究随机测试故障检测能力的基础

- $r = |F| / |S|$
 - S 为待测系统的所有可能输入
 - F 为触发缺陷的测试输入 (S 的子集)
- 在均匀采样方式下, 一个随机选择的测试输入触发缺陷的概率 (uniform distribution)

随机测试的缺陷检测能力

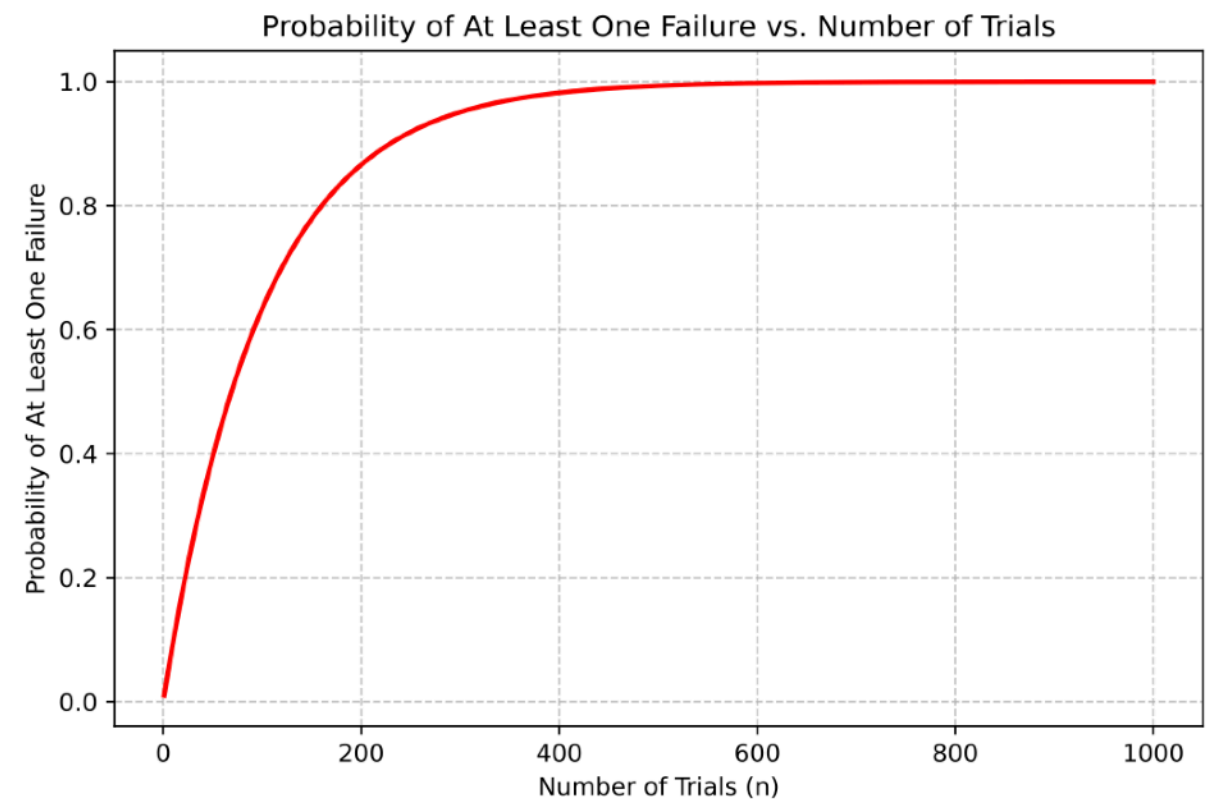
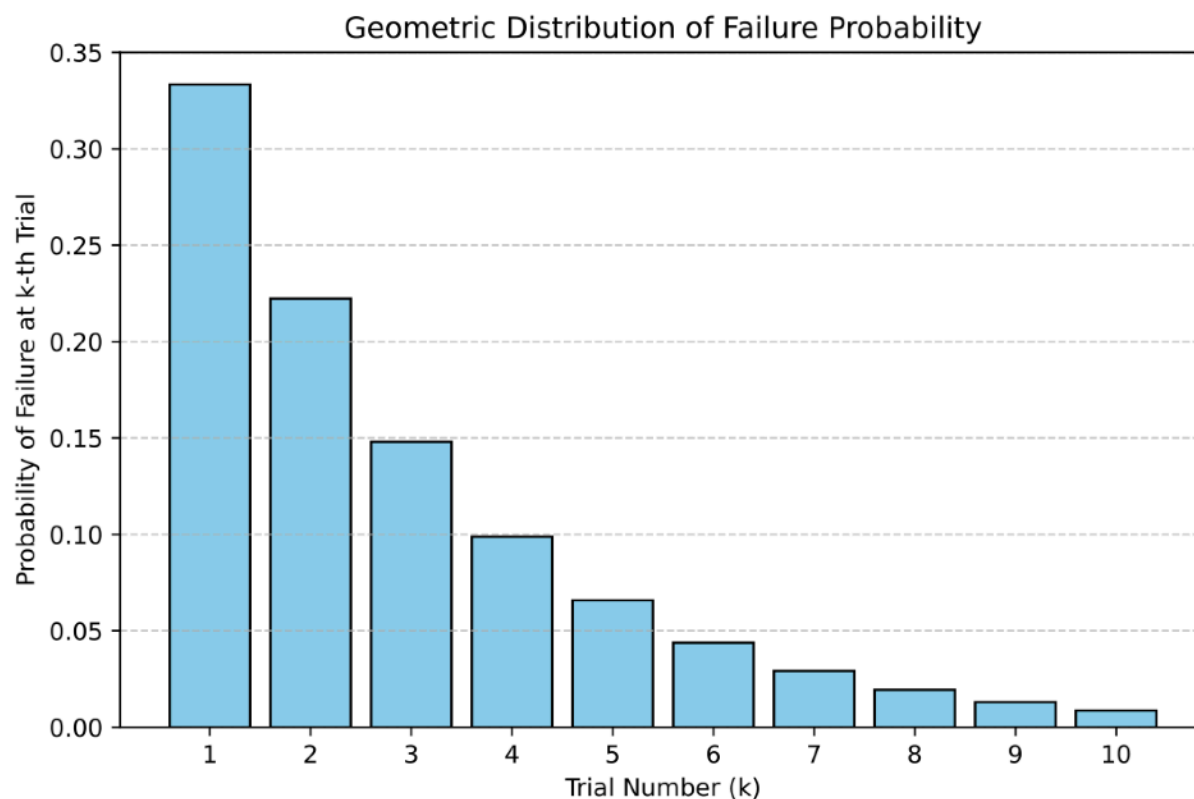
有了失效率之后，我们就可以回答

- 在执行到第 k 个测试用例时才发现 Failure 的概率

$$(1 - r)^{(k-1)} r$$

- 如果执行了 k 个测试用例，则至少能找到一个 Failure 的概率

$$1 - (1 - r)^k$$



随机测试的缺陷检测能力

- 如何知道一个具体待测软件的失效率？
 - 根据历史经验 / 待测软件类型等进行估计
 - 提供一种理论分析的视角
- 如果我们对待测系统及其故障有某种深刻的认识？
 - 以更大概率采样特定的测试输入
 - 例如，更大的数值、更长的数组等

构造随机测试输入

生成一个随机的 Number 很简单，但对于复杂数据结构就没那么容易

- 如何构造一个随机的 Tree / Graph / PDF File / Program / Image ...?
- 不论是什么数据结构，底层存储都是 0 和 1 的 Bit Sequence
 - 随机生成 Random Bits 就可以了
 - 但 Random Trees 并不是 Random Bits ➡ 输入结构/语义的合法性
- 另一种尝试是以增量方式构造数据结构
 - 在每个决策点都做随机的选择
 - 但可能会导致偏向特定类型的测试输入

构造随机测试输入

测试输入的长度也会对 Randomness 产生影响

- 假设我们可以构造长度最长为 L 的 Binary String
- 如果在整个测试输入空间中均匀采样
 - 因为 $2^L > 2^{L-1} + \dots + 2^2 + 2^1$
 - 所以会有更大概率选择到长度为 L 的测试输入
- 另外一种策略
 - 先随机确定输入的长度
 - 然后再随机构造满足该长度的 String

构造随机测试输入

如果测试对象是 Class (Object-Oriented Programs)

```
public class HashSet extends Set {  
    public boolean add(Object o) { ... }  
    public boolean remove(Object o) { ... }  
    public boolean isEmpty() { ... }  
    public boolean equals(Object o) { ... }  
    ...  
}
```

构造随机测试输入

如果测试对象是 Class (Object-Oriented Programs)

```
Set s = new HashSet();  
s.add("hi");
```

```
Set s = new HashSet();  
s.add("hi");  
s.remove(null);
```

```
Set s = new HashSet();  
s.isEmpty();  
s.remove("no");  
s.isEmpty();  
s.add("no");  
s.isEmpty();  
s.isEmpty();
```

构造一个随机的方法调用序列，每个调用涉及

- 一个随机选择的 method
- 一些随机选择的 parameter values
- 一个随机选择的 receiver object (if any)

Feedback-directed Random Test Generation

Carlos Pacheco¹, Shuvendu K. Lahiri², Michael D. Ernst¹, and Thomas Ball²

¹MIT CSAIL, ²Microsoft Research

{cpacheco, mernst}@csail.mit.edu, {shuvendu, tball}@microsoft.com

Abstract

We present a technique that improves random test generation by incorporating feedback obtained from executing test inputs as they are created. Our technique builds inputs incrementally by randomly selecting a method call to apply and finding arguments from among previously-constructed inputs. As soon as an input is built, it is executed and checked against a set of contracts and filters. The result of the execution determines whether the input is redundant, illegal, contract-violating, or useful for generating more inputs. The technique outputs a test suite consisting of unit tests for the classes under test. Passing tests can be used to ensure that code contracts are preserved across program changes; failing tests (that violate one or more contract) point to potential errors that should be corrected.

Our experimental results indicate that feedback-directed random test generation can outperform systematic and undirected random test generation, in terms of coverage and error detection. On four small but nontrivial data structures (used previously in the literature), our technique achieves higher or equal block and predicate coverage than model checking (with and without abstraction) and undirected random generation. On 14 large, widely-used libraries (comprising 780KLOC), feedback-directed random test generation finds many previously-unknown errors, not found by either model checking or undirected random generation.

1 Introduction

There is an ongoing controversy regarding the relative merits of random testing and systematic testing. Theoretical work suggests that random testing is as effective as systematic techniques [8, 15]. However, some believe that in practice, random testing cannot be as effective as systematic testing because many interesting tests have very little chance of being created at random. Previous empirical studies [9, 18, 28] found that random test input generation achieves less code coverage than systematic genera-

tion [18], model checking, and symbolic execution [28].

It is difficult to generalize the results of these studies with regard to the relative advantages of random and systematic testing. The evaluations were performed on very small programs. Because the small programs apparently contained no errors, the comparison was in terms of coverage or rate of mutant killing [21], not in terms of true error detection, which is the best measure to evaluate test input generation techniques. While the systematic techniques used sophisticated heuristics to make them more effective, the type of random testing used for comparison is unguided random testing, with no heuristics to guide its search.

Our work addresses random generation of unit tests for object-oriented programs. Such a test typically consists of a sequence of method calls that create and mutate objects, plus an assertion about the result of a final method call. A test can be built up iteratively by randomly selecting a method or constructor to invoke, using previously-computed values as inputs. It is only sensible to build upon a legal sequence of method calls, each of whose intermediate objects is sensible and none of whose methods throw an exception indicating a problem. For example, if the one-method test `a=sqrt(-1)` is erroneous (say, the argument is required to be non-negative), then there is no sense in building upon it to create the two-method test `a=sqrt(-1); b=log(a)`. Our technique uses feedback obtained from executing the sequence as it is being constructed, in order to guide the search toward sequences that yield *new* and *legal* object states. Inputs that create redundant or illegal states are never extended; this has the effect of pruning the search space.

We have implemented the technique in RANDOOP.¹ RANDOOP is fully automatic, requires no input from the user (other than the name of a binary for .NET or a class directory for Java), and scales to realistic applications with hundreds of classes. RANDOOP has found serious errors in widely-deployed commercial and open-source software.

Figure 1 shows a test case generated by RANDOOP when run on Sun's JDK 1.5. The test case shows a violation of the `equals` contract: a set `s1` returned by `unmodifiable-`

Pacheco C, Lahiri S K, Ernst M D, et al. Feedback-directed random test generation. International Conference on Software Engineering (ICSE), 2007: 75-84.



Randoop

Automatic unit test generation for Java

Download the
Latest Release

View the
GitHub Project

What is Randoop?

Randoop is a unit test generator for Java. It automatically creates unit tests for your classes, in JUnit format.

The [Randoop manual](#) tells you how to [install](#) and [run](#) Randoop.

How does Randoop work?

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 can be used for two purposes: to find bugs in your program, and to create regression tests to warn you if you change your program's behavior in the future.

Randoop's combination of test generation and test execution results in a highly effective test generation technique. Randoop has revealed previously-unknown errors even in widely-used libraries including Sun's and IBM's JDKs and a core .NET component. Randoop continues to be used in industry, for example at ABB corporation.

Documentation

[Randoop user manual](#)

For Randoop developers/contributors:

Randoop

Feedback-Directed Random Test Generation

一种增量式的构造方法

- 在已有测试数据 (方法调用序列) 的基础上进行扩展以生成新的测试数据
- 使用当前测试执行结果来指导后续测试数据的生成
 - 避免生成冗余和无效的测试数据
 - 尝试覆盖尽可能多的对象状态

Randoop

Feedback-Directed Random Test Generation

Create a value pool for primitive type $\leftarrow \{true, false, 0, 1, "hi", null, \dots\}$

Repeat until time limit expires:

- **Randomly** pick a public method $m(T_1, \dots, T_k) \rightarrow T_{ret}$
- For each parameter of type T_i , **randomly** pick a sequence S_i from the value pool that constructs an object v_i of type T_i
- Create a new sequence $S_{new} = S_1 ; \dots ; S_k ; T_{ret} \ v_{new} = m(v_1, \dots, v_k)$
- If S_{new} was previously created, discard it
- Execute S_{new} (check the contracts after each method call) and add it into the pool

Randoop

Program Under Test

```
public class A{  
    public A() {...}  
    public B m1(A a) {...}  
}  
public class B{  
    public B(int i) {...}  
    public void m2(B b, A a) {...}  
}
```

Sequence Pool

Randoop

Program Under Test

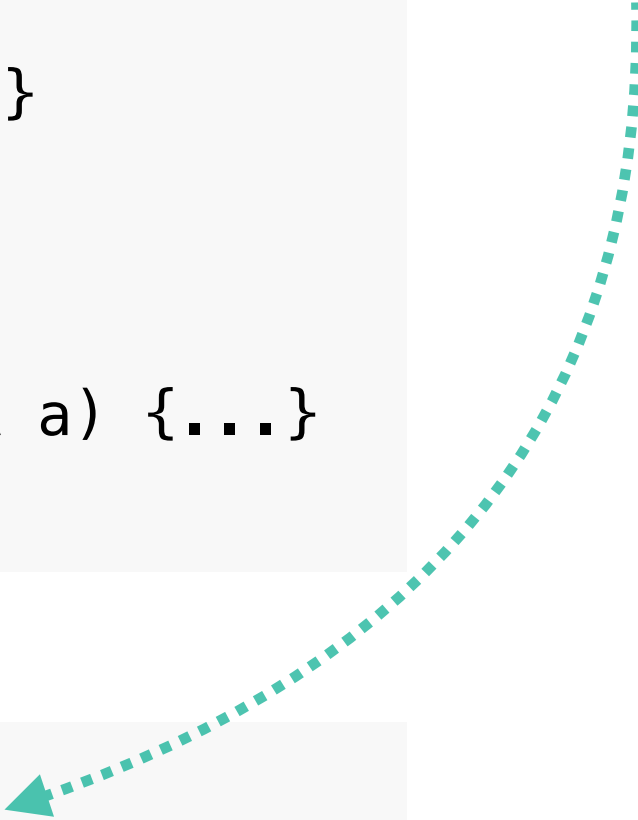
```
public class A{  
    public A() {...}  
    public B m1(A a) {...}  
}  
public class B{  
    public B(int i) {...}  
    public void m2(B b, A a) {...}  
}
```

Test I

```
B b1 = new B(0);
```

Sequence Pool

```
s1: B b1 = new B(0);
```



Randoop

Program Under Test

```
public class A{  
    public A() {...}  
    public B m1(A a) {...}  
}  
public class B{  
    public B(int i) {...}  
    public void m2(B b, A a) {...}  
}
```

Test 1

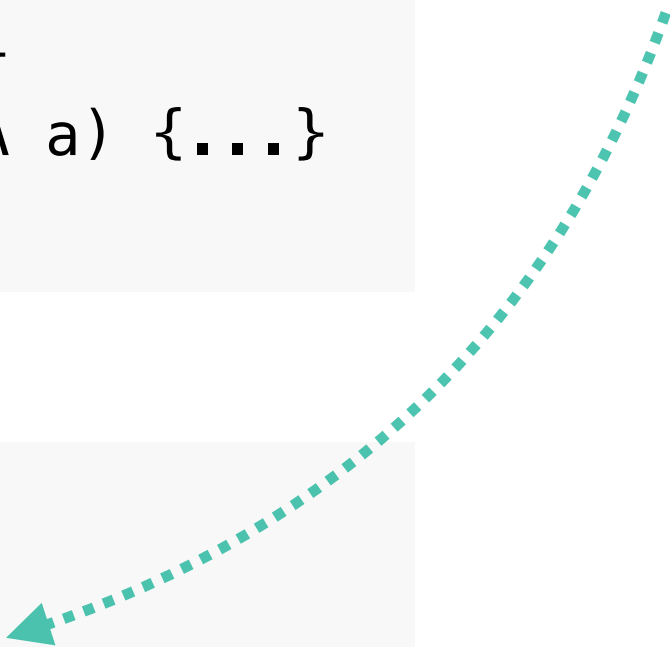
```
B b1 = new B(0);
```

Test 2

```
A a1 = new A();
```

Sequence Pool

```
s1: B b1 = new B(0);  
s2: A a1 = new A();
```



Randoop

Program Under Test

```
public class A{  
    public A() {...}  
    public B m1(A a) {...}  
}  
public class B{  
    public B(int i) {...}  
    public void m2(B b, A a) {...}  
}
```

Sequence Pool

```
s1: B b1 = new B(0);  
s2: A a1 = new A();  
s3: A a1 = new A();  
    B b2 = a1.m1(a1);
```

Test 1

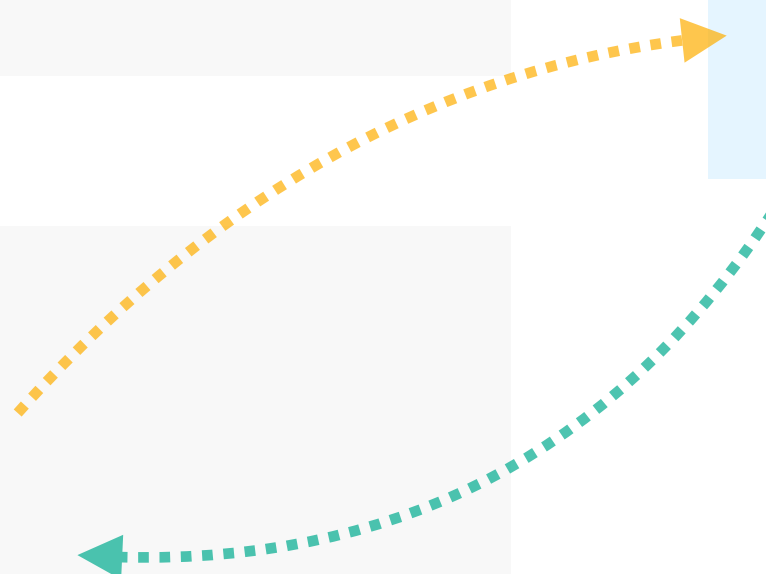
```
B b1 = new B(0);
```

Test 2

```
A a1 = new A();
```

Test 3

```
A a1 = new A();    // from s2  
B b2 = a1.m1(a1);
```



Randoop

Program Under Test

```
public class A{  
    public A() {...}  
    public B m1(A a) {...}  
}  
public class B{  
    public B(int i) {...}  
    public void m2(B b, A a) {...}  
}
```

Sequence Pool

```
s1: B b1 = new B(0);  
s2: A a1 = new A();  
s3: A a1 = new A();  
    B b2 = a1.m1(a1);  
s4: ...
```

Test 1

```
B b1 = new B(0);
```

Test 2

```
A a1 = new A();
```

Test 3

```
A a1 = new A();    // from s2  
B b2 = a1.m1(a1);
```

Test 4

```
B b1 = new B(0);    // from s1  
A a1 = new A();    // from s3  
B b2 = a1.m1(a1);  
b1.m2(b2, a1);
```

Randoop

Feedback-Directed Random Test Generation

使用代码应满足的基本性质 (Code Contracts) 作为 Test Oracle

- 对于 Method
 - 不应该抛出 `AssertionError`
 - 如果参数不是 *Null* 的话不应该抛出 `NullPointerException`
- 对于 Object
 - `o.equals(o)` 应该为 *True*
 - `o.equals(o)` 不应该抛出异常
 - `o.hashCode()` 不应该抛出异常
 - `o.toString()` 不应该抛出异常

Randoop

Feedback-Directed Random Test Generation

在工具实现上支持 Regression Test 和 Error-Revealing Test 的生成

- Regression Test (detect *future* bugs)
 - 记录程序当前行为，用于后续回归测试
- Error-Revealing Test (detect *current* bugs)
 - 揭示当前程序可能存在的缺陷
 - 支持对 Error-Revealing Test 进行 Minimisation

Randoop

Feedback-Directed Random Test Generation

曾发现 OpenJDK 中的真实故障

@Test

```
public static void test1() {  
    LinkedList list = new LinkedList();  
    Object o1 = new Object();  
    list.addFirst(o1);  
  
    // A TreeSet is an ordered collection, and this call should  
    // throw a ClassCastException because the list element is not  
    // Comparable. But the constructor silently accepts the list  
    TreeSet t1 = new TreeSet(list);  
  
    Set s1 = Collections.synchronizedSet(t1);  
  
    // At this point, we created a set (s1) that violations reflexivity  
    // of equality: it is not equal to itself!  
    org.junit.Assert.assertEquals(s1, s1);  
}
```

改进随机测试

假设我们要测试 `void foo(int n)`

```
// Test Suite A
```

```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10
```

```
// Test Suite B
```

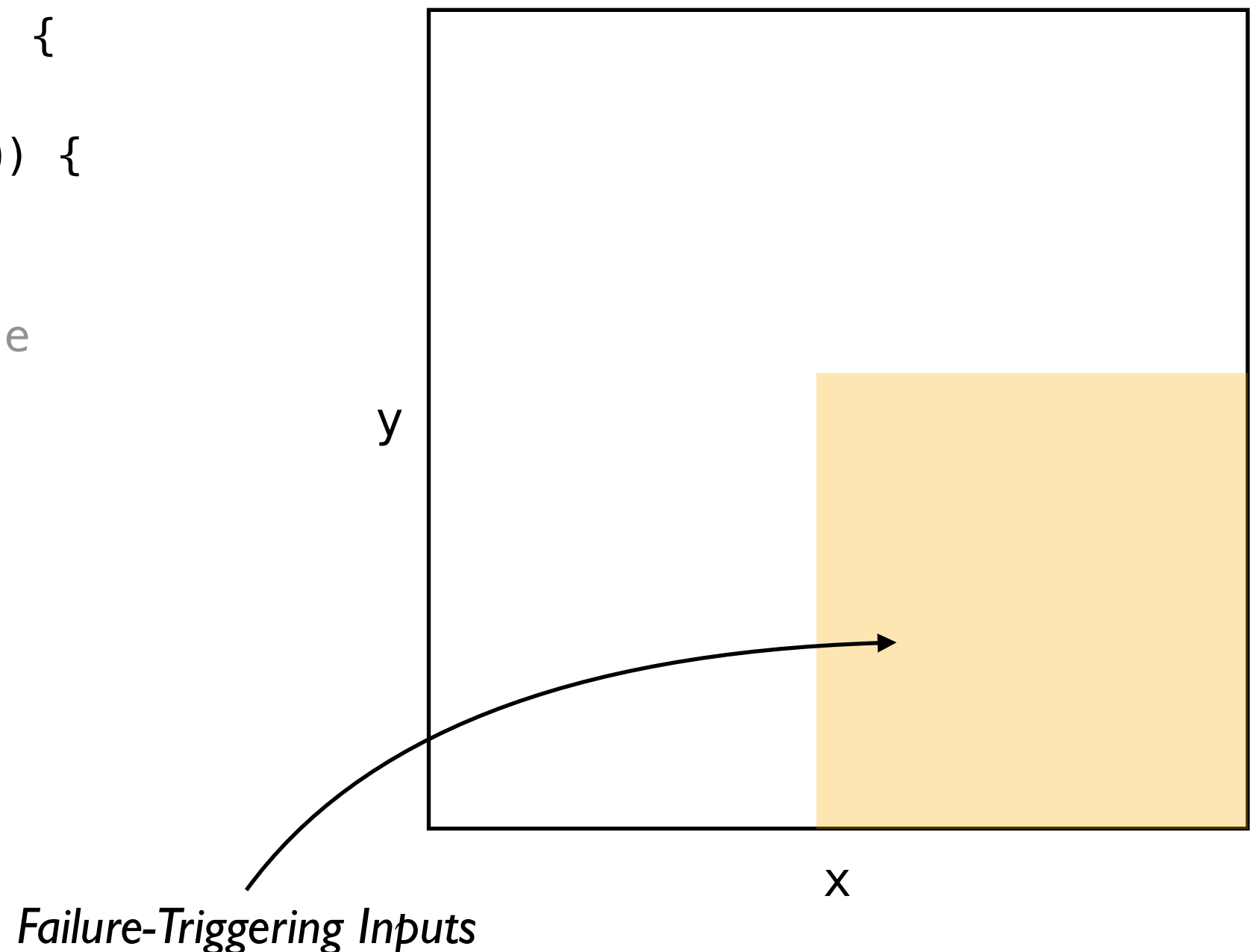
```
-2345, 12, 342, -4443, 2, 9437, -18, 24, 9934332, -5244
```

- 那一组测试数据更好?
- 在没有任何额外假设的情况下，它们在故障检测能力上没有差别

改进随机测试

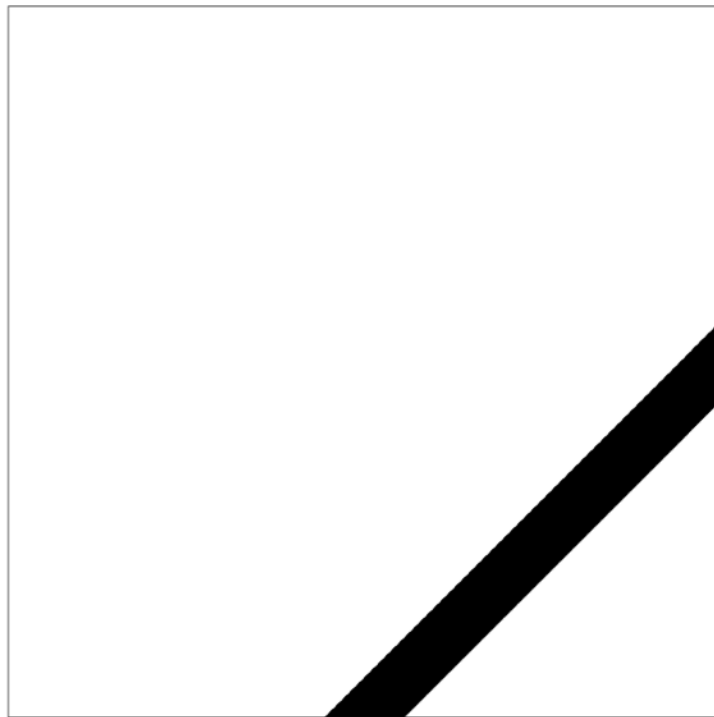
如果我们对测试输入空间中的 Failure Region 做些假设

```
int foo(int x, int y) {  
    // do something  
    if (x > 0 && y < 10) {  
        // faulty code  
    }  
    // do something else  
}
```

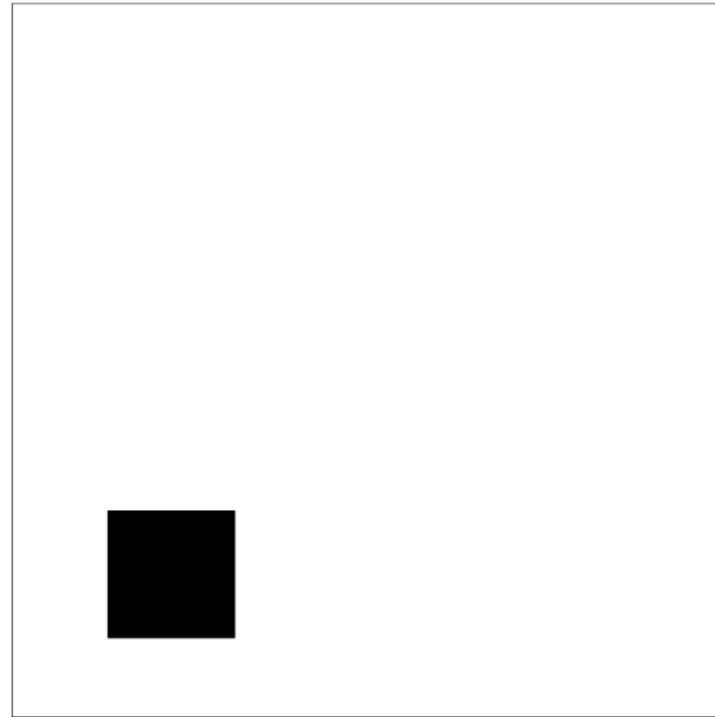


改进随机测试

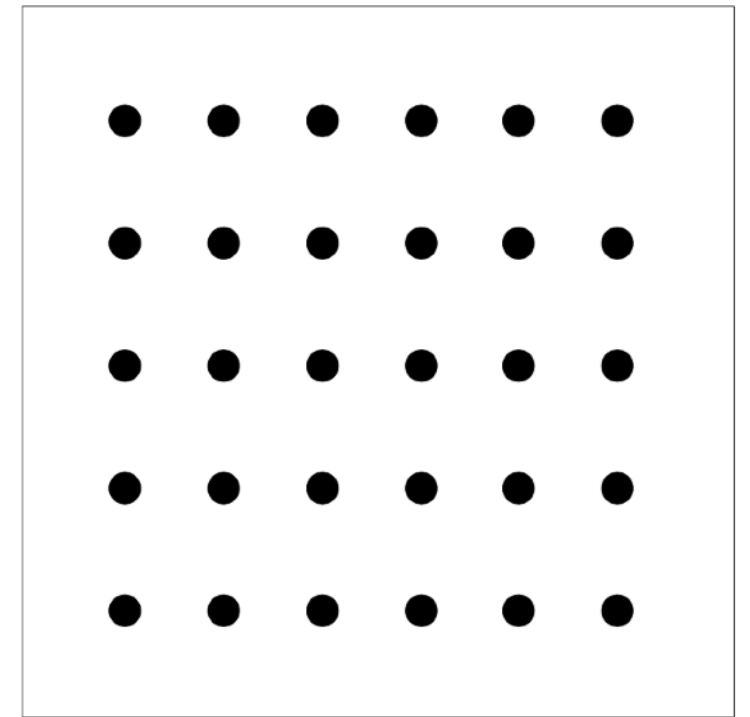
如果我们对测试输入空间中的 Failure Region 做些假设



Strip Pattern



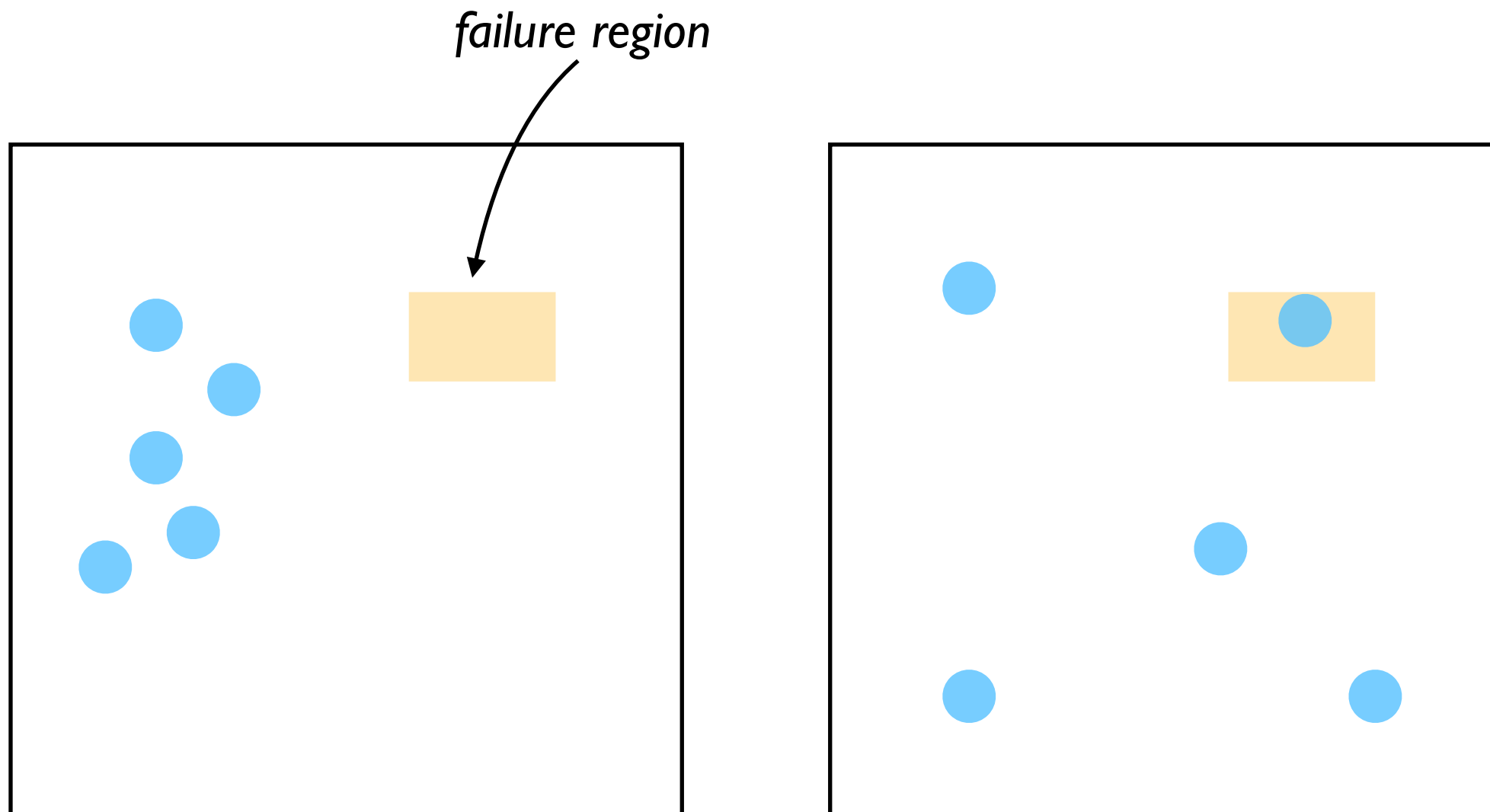
Block Pattern



Point Pattern

改进随机测试

如果我们对测试输入空间中的 Failure Region 做些假设



自适应随机测试

- 🤔 使用所有可能的输入进行测试是不可行的
- 🤔 触发缺陷的测试输入 (failure-triggering inputs) 通常会在测试输入空间中聚集 (cluster)
- 💡 让测试输入数据尽可能均匀分布 (evenly spread)

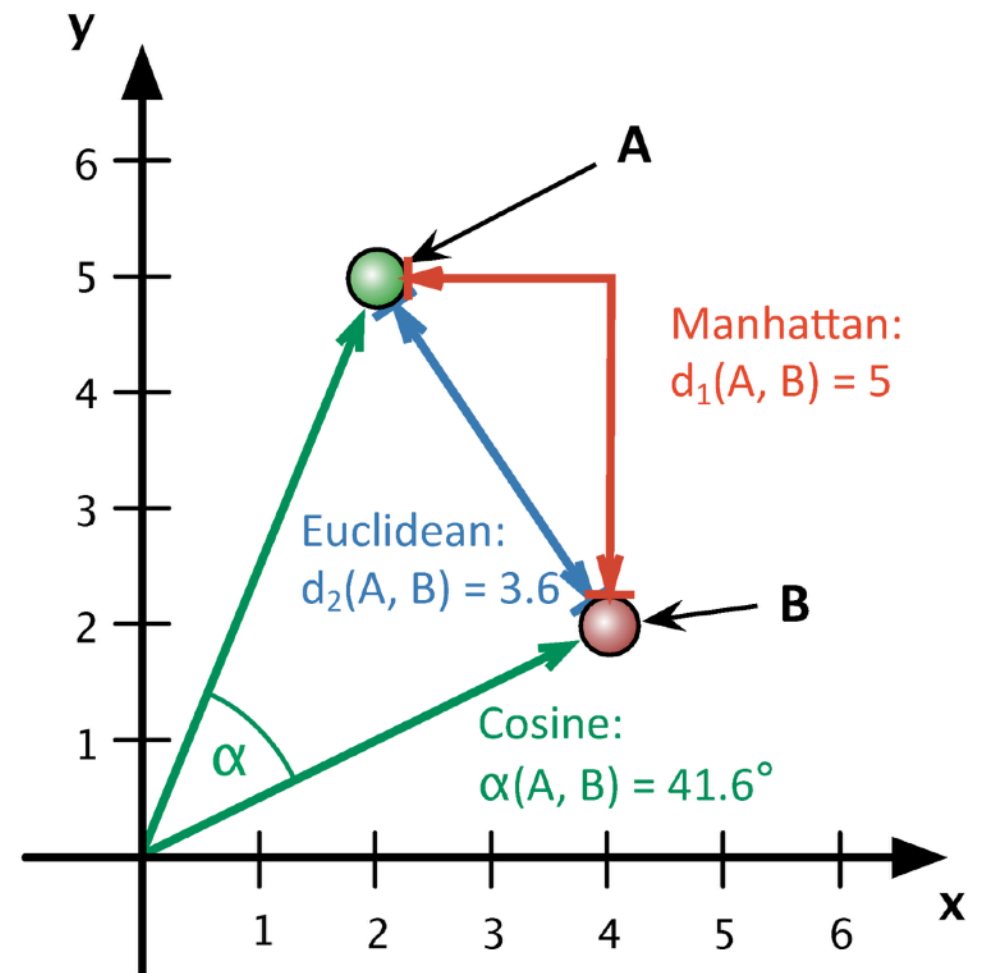
距离度量

基于距离度量实现均匀分布：最大化测试用例之间的距离

- 对于数值类的输入，通常可以使用
 - Euclidean Distance
 - Manhattan Distance
 - Cosine Distance
- 对于字符串类的输入，可以使用 Hamming Distance (Edit Distance)
 - 将一个字符串更改为另一个字符串所需的编辑次数

$$\text{dist}(\text{ABC}\text{A}, \text{AC}\text{CAB}) = 2$$

- 为待测程序设计一个好的距离度量是一个挑战



FSCS-ART

Fixed-Size-Candidate-Set

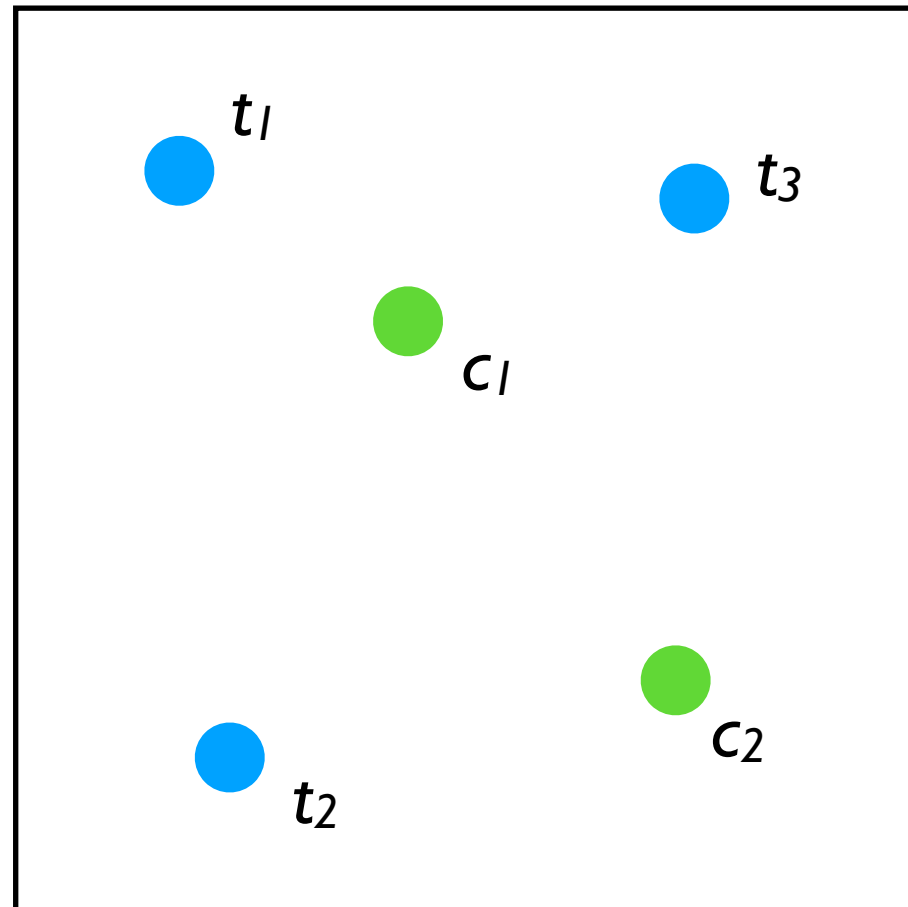
$T = \{\text{test inputs that have been executed}\}$ (executed set)

Generate a new test input at each time:

1. $C =$ sample a set of N inputs at random (candidate set)
2. Choose c from C so that it is the farthest away from all the inputs in T , that is, $\forall c' \in C, \min_{t \in T} \text{dist}(c, t) \geq \min_{t \in T} \text{dist}(c', t)$
3. Add c into T

FSCS-ART

Fixed-Size-Candidate-Set

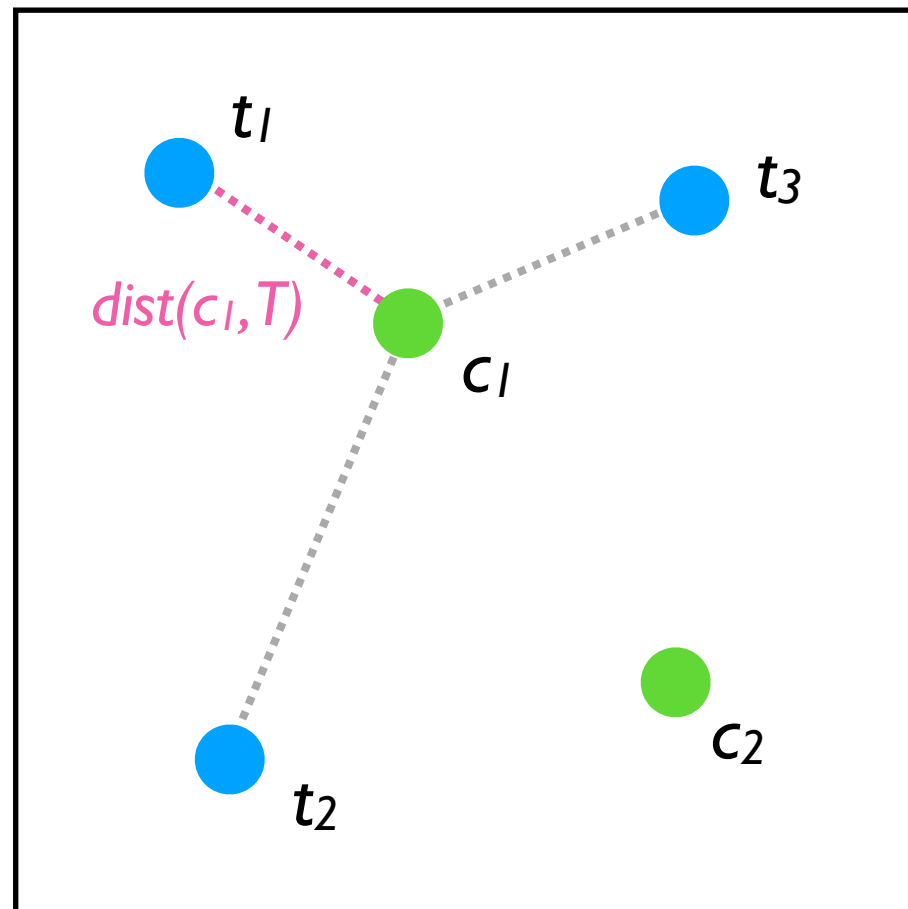


$T = \{t_1, t_2, t_3\}$ (executed set)

$C = \{c_1, c_2\}$ (candidate set)

FSCS-ART

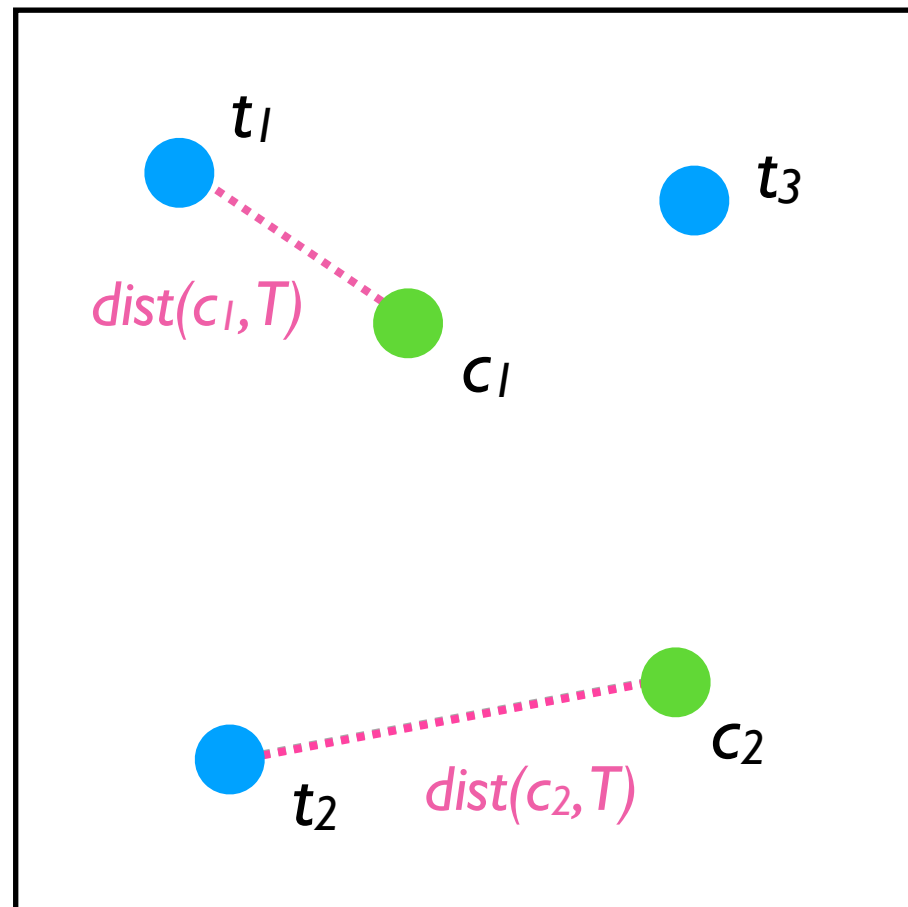
Fixed-Size-Candidate-Set



计算 c_1 和 T 中每个测试用例的距离，
并使用其中的最小值作为 c_1 和 T 的
距离 $dist(c_1, T)$

FSCS-ART

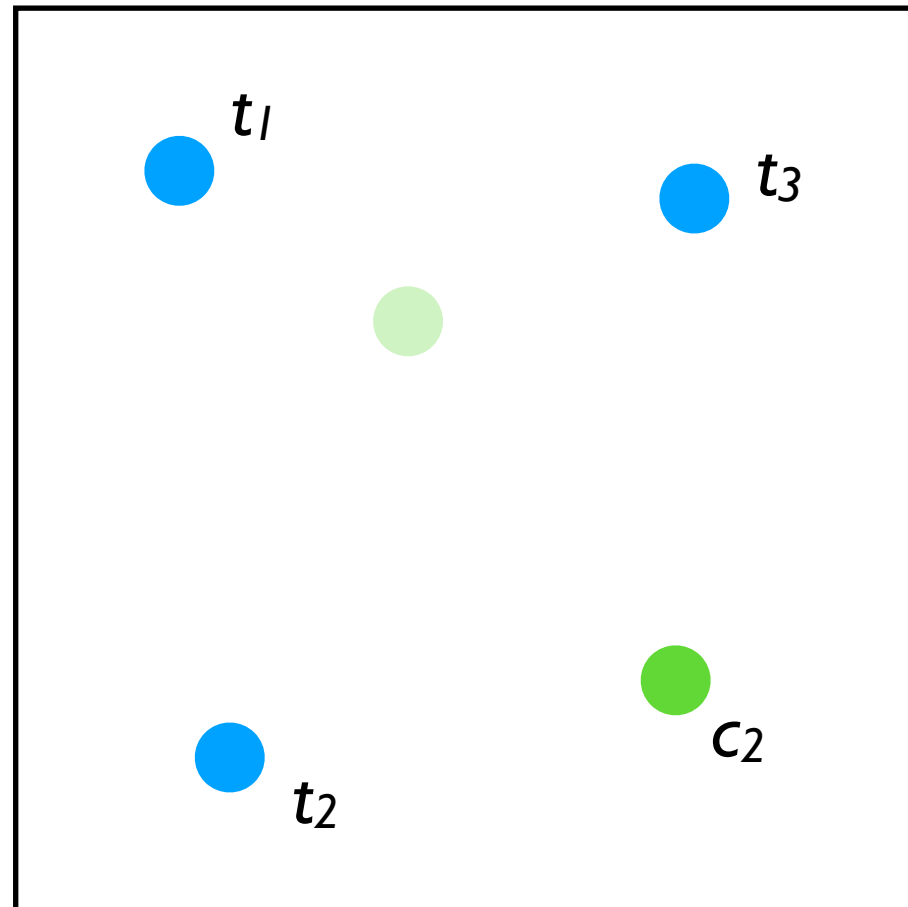
Fixed-Size-Candidate-Set



类似地，计算 c_2 和 T 的距离 $dist(c_2, T)$

FSCS-ART

Fixed-Size-Candidate-Set



选择和 T 距离最远的测试用例
(一种 *max-min distance* 策略)

对均匀分布程度的度量

给定一个测试用例集，评估其中测试用例的均匀分布程度

- **Discrepancy:** 不同 subdomain 中测试用例的数量 T_i 是否与该 subdomain 相对整个测试输入空间的大小 D_i 成比例

$$Discrepancy(T) = \max_{1 \leq i \leq m} \left| \frac{|T_i|}{|T|} - \frac{|D_i|}{|D|} \right|$$

- **Diversity:** 每个测试用例与其余测试用例间距离的和

$$Diversity(T) = \sum_{1 \leq i \leq n} \min_{1 \leq j \neq i \leq n} dist(t_i, t_j)$$

ART 的故障检测能力

考虑触发第一个失效所需的测试用例个数 (F-measure)

- 假设 Failure Rate 为 r
 - 使用 Uniform Sampling 的 RT: $F\text{-measure} = 1 / r$
 - 使用 ART (基于已有研究): $F\text{-measure} \approx 1 / 2r$

Adaptive Random Testing: An Illusion of Effectiveness?

Andrea Arcuri
Simula Research Laboratory
P.O. Box 134, 1325 Lysaker, Norway
arcuri@simula.no

Lionel Briand
Simula Research Laboratory and
University of Oslo
P.O. Box 134, 1325 Lysaker, Norway
briand@simula.no

ABSTRACT

Adaptive Random Testing (ART) has been proposed as an enhancement to random testing, based on assumptions on how failing test cases are distributed in the input domain. The main assumption is that failing test cases are usually grouped into contiguous regions. Many papers have been published in which ART has been described as an effective alternative to random testing when using the average number of test case executions needed to find a failure (F-measure). But all the work in the literature is based either on simulations or case studies with unreasonably high failure rates. In this paper, we report on the largest empirical analysis of ART in the literature, in which 3727 mutated programs and nearly ten trillion test cases were used. Results show that ART is highly inefficient even on trivial problems when accounting for distance calculations among test cases, to an extent that probably prevents its practical use in most situations. For example, on the infamous Triangle Classification program, random testing finds failures in few milliseconds whereas ART execution time is prohibitive. Even when assuming a small, fixed size test set and looking at the probability of failure (P-measure), ART only fares slightly better than random testing, which is not sufficient to make it applicable in realistic conditions. We provide precise explanations of this phenomenon based on rigorous empirical analyses. For the simpler case of single-dimension input domains, we also perform formal analyses to support our claim that ART is of little use in most situations, unless drastic enhancements are developed. Such analyses help us explain some of the empirical results and identify the components of ART that need to be improved to make it a viable option in practice.

Categories and Subject Descriptors

D.2.5 [Software Engineering]: Testing and Debugging;
I.2.8 [Artificial Intelligence]: Problem Solving, Control Methods, and Search

General Terms

Algorithms, Experimentation, Reliability, Theory

Keywords

Faulty region, random testing, F-measure, P-measure, shape, diversity, similarity, distance.

1. INTRODUCTION

Random testing does not exploit any information about the software under test (SUT). Therefore, historically it was considered a naive testing strategy that is less effective than partition testing. Myers in his 1979 book [34] states “probably the poorest [test case design] methodology is random input testing ...”. But this statement was not agreed upon by everyone. Thayer *et al.* [41] for example were favorable to random testing, recommending it as a necessary final step of the testing activity.

In a seminal paper, Duran and Ntafos [17] carried out a series of experiments in which they showed that random testing could be more effective than the commonly used partition testing. That was a counterintuitive result that opened the doors to a large body of literature on the properties and benefits of random testing. Random testing has been shown to be a very useful tool in the hands of software testers and it is not a minor technique as it was originally thought [17, 5].

Usually, in random testing the test cases are sampled at random from the input domain D of the SUT with uniform probability. In other words, each test case is sampled with probability $p = 1/|D|$. The motivation is that, if we do not know anything about the SUT, then each test case is as likely as the others to be able to detect failures. However, if the distribution of failing test cases follows some specific patterns, then this information could be exploited to give higher probability to sampling test cases that are more likely to detect faults.

One property that has been noticed in numerical applications is that often failing test cases tend to cluster in *contiguous regions* of the input domain D [42, 1, 19, 38], referred to as *faulty regions*. For example, in a function that takes as input two integers x and y , if there is a fault inside a conditional statement such as `if (x >= 0 && y >= 0 && x >= y)`, then we could have a contiguous region of failing test cases in the positive sub-area $x \geq y$.

Based on the above assumption, a modified version of random testing has been proposed: Adaptive Random Testing (ART) [13, 12]. The fundamental idea behind ART is to reward *diversity* among sampled test cases. If a test case does not detect any failure, then in presence of contiguous faulty regions it would be better to sample

Arcuri A, and Briand L. Adaptive random testing: An illusion of effectiveness? International Symposium on Software Testing and Analysis (ISSTA), 2011: 265-275.

ART 的故障检测能力

除了测试用例数量之外，如果进一步考虑测试用例的生成成本？
(test case generation cost)

- 当候选测试用例集大小为 N 时，使用 FSCS-ART 生成 K 个测试用例总共需要的距离计算次数：

$$0 + N + 2N + 3N + \dots + (K - 1) * N = N * \frac{K * (K - 1)}{2}$$

- 选择一个好的测试用例是有 "代价" 的
 - 把 CPU 时间花在距离计算上，还是花在生成更多的随机测试用例
 - 随着 Failure Rate 降低 (需要更大的 K)，和 RT 的成本差异会进一步扩大
- 如果进一步考虑测试用例的执行开销？
(test case execution cost)

ART 的故障检测能力

更远的距离不一定总是意味着更强的故障检测能力

Assume a 1-dimensional integer domain D

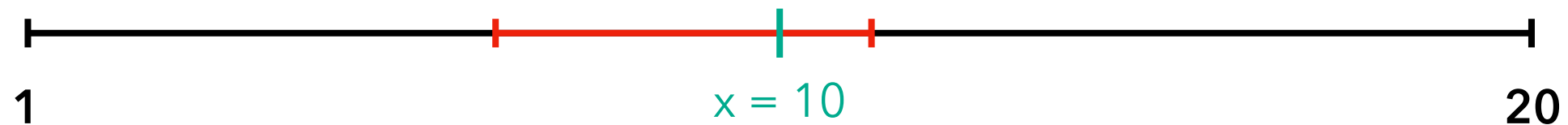


There will be $|D| - z + 1$ ways of placing the failure region in D

ART 的故障检测能力

更远的距离不一定总是意味着更强的故障检测能力

Assume a 1-dimensional integer domain D



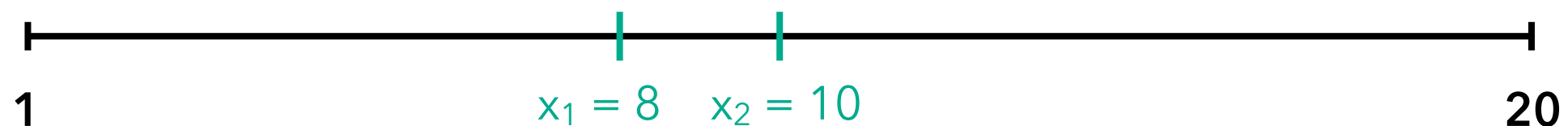
This test suite can cover $z = 5$ failure region placements

$[6, 10], [7, 11], [8, 12], [9, 13], [10, 14]$

ART 的故障检测能力

更远的距离不一定总是意味着更强的故障检测能力

Assume a 1-dimensional integer domain D



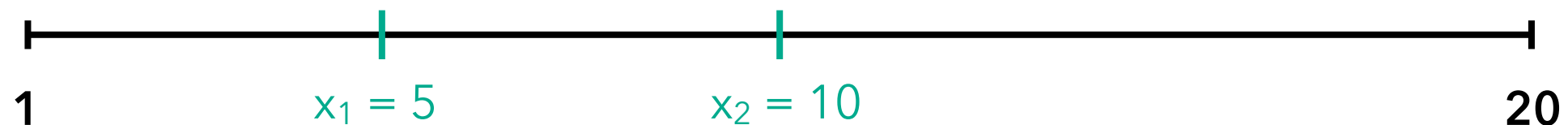
This test suite can cover 7 failure region placements

$[4, 8], [5, 9], [6, 10], [7, 11], [8, 12], [9, 13], [10, 14]$

ART 的故障检测能力

更远的距离不一定总是意味着更强的故障检测能力

Assume a 1-dimensional integer domain D



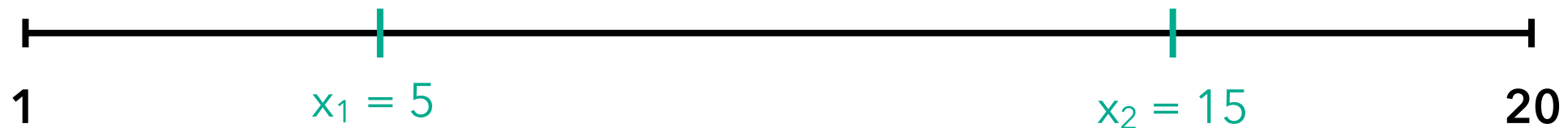
This test suite can cover $2z = 10$ failure region placements

[1, 5], [2, 6], [3, 7], [4, 8], [5, 9]
[6, 10], [7, 11], [8, 12], [9, 13], [10, 14]

ART 的故障检测能力

更远的距离不一定总是意味着更强的故障检测能力

Assume a 1-dimensional integer domain D



Further increase the distance between the two test cases will not necessarily increase their fault detection capability

ART 的故障检测能力

更远的距离不一定总是意味着更强的故障检测能力

Theorem. For the set of programs S , assuming one faulty region of size z , a sufficient condition to maximise the probability that a set K of test cases detects a failure is when

$$\min(\text{dist}(x_{\min}, x_1), \text{dist}(x_i, x_{i+1}), \text{dist}(x_{|K|}, x_{\max})) \geq z .$$

- 在超过一定的阈值之后，继续增大测试用例间距离未必带来收益
- 当测试输入空间很大且测试用例数量较少时，RT 生成的测试用例也很有可能满足上述性质 (随机采样已经足够 "均匀分布")
- 但如果是 **non-numeric input space** 和 **non-contiguous failure region** ?



Adaptive Random Testing: The ART of test case diversity

Tsong Yueh Chen^{a,*}, Fei-Ching Kuo^a, Robert G. Merkel^a, T.H. Tse^b

^a Faculty of Information and Communication Technologies, Swinburne University of Technology, John St, Hawthorn 3122, Australia

^b Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong

ARTICLE INFO

Article history:

Received 7 November 2008

Accepted 13 February 2009

Available online 4 March 2009

Keywords:

Software testing

Random testing

Adaptive random testing

Adaptive random sequence

Failure-based testing

Failure pattern

ABSTRACT

Random testing is not only a useful testing technique in itself, but also plays a core role in many other testing methods. Hence, any significant improvement to random testing has an impact throughout the software testing community. Recently, *Adaptive Random Testing* (ART) was proposed as an effective alternative to random testing. This paper presents a synthesis of the most important research results related to ART. In the course of our research and through further reflection, we have realised how the techniques and concepts of ART can be applied in a much broader context, which we present here. We believe such ideas can be applied in a variety of areas of software testing, and even beyond software testing. Amongst these ideas, we particularly note the fundamental role of diversity in test case selection strategies. We hope this paper serves to provoke further discussions and investigations of these ideas.

© 2009 Elsevier Inc. All rights reserved.

1. Introduction

Despite decades of effort to develop alternative technologies, software testing remains the primary way to verify the quality of software systems. However, it remains a labour-intensive, slow and imperfect process. It is, therefore, important to consider how testing can be performed more effectively and at a lower cost through the use of systematic, automated methods.

Attempts to automate the generation of test data through various forms of random selection date from the early 1960s (Renfer, 1962), and have been a regular feature of the research literature and industrial practice. Random testing is simple in concept, often easy to implement, has been demonstrated to effectively detect failures, is good at exercising systems in unexpected ways (which may not occur to a human tester), and may be the only practical choice when the source code and the specifications are unavailable or incomplete. It has been used extensively as a testing method in itself; furthermore, it forms a core part of many other testing methods. On the other hand, it is often argued that such random testing is inefficient, as there is no attempt to make use of any available information to guide testing. This work now encompasses

result in failures in contiguous areas of the input domain, known as *failure patterns*. ART systematically guides, or filters, randomly generated candidates, to take advantage of the likely presence of such patterns. In this paper, we attempt to provide a synthesis of the growing body of piecemeal research in the area of ART. We provide new insights and interpretations by explicitly drawing out novel links among individual research results. We report how ART can be adapted for testing a wide variety of types of software, and how it has inspired theoretical analysis that has revealed fundamental connections between software failure behaviour and maximum testing effectiveness. We believe that this work, viewed as a whole, is significant both to researchers in software testing and to practitioners.

Section 2 introduces the concept of failure patterns, which provided the key inspiration for ART, the fundamentals of which are described in Section 3. Under certain conditions, ART is close to an optimally effective test case selection strategy, as we outline in Section 4. Section 5 describes how ART can be applied to a wide range of software through the development of appropriate “distance measures”. In Section 6, our attention turns from summarising work already conducted to exploring possibilities for



Tsong Yueh Chen
陈宗岳

Chen T Y, Kuo F C, Merkel R G, et al. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software* (JSS), 2010, 83(1): 60-66.

通过 Diversity 提升测试的有效性

Test Input Sampling (Selection) is all about *Diversity*

- 距离度量只是 Diversity 的一种表现形式
- ART 是一系列旨在提高 Diversity 的测试数据生成算法
 - FSCS-ART 是其中一种最为经典的实现

PB-ART

Partition Based-ART

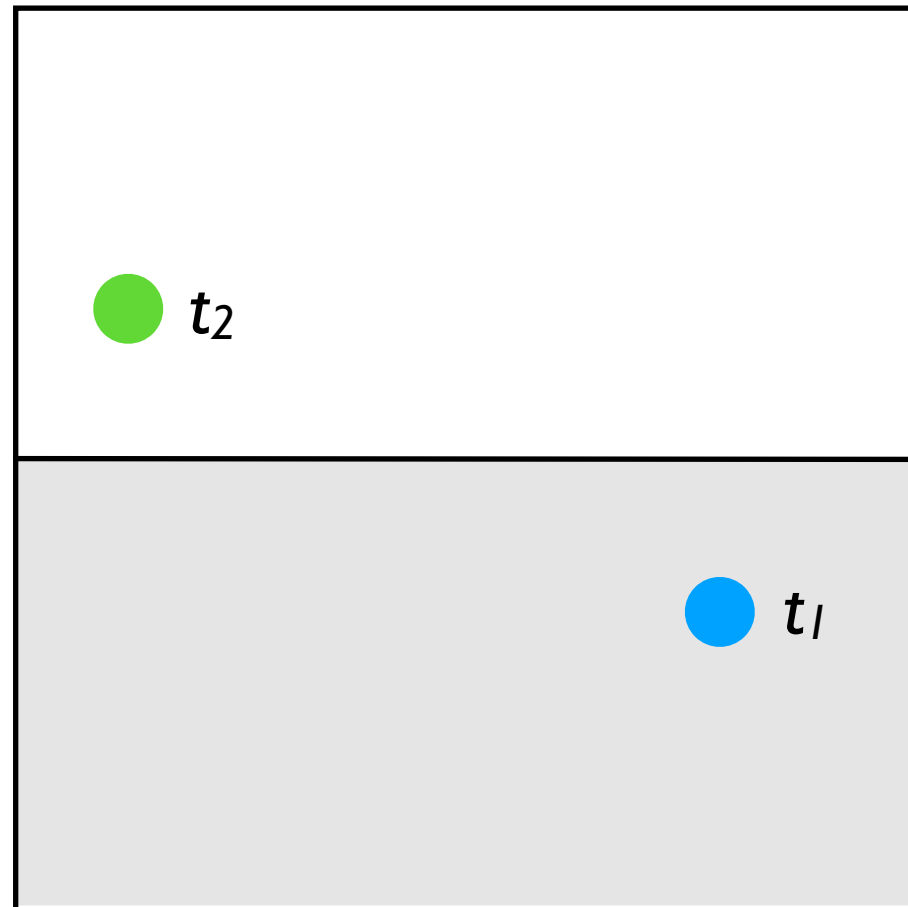
$T = \{\text{test inputs that have been executed}\}$

Generate a new test input at each time:

1. Partition the input domain D into m disjoint subdomains according to a given criterion
2. Choose a subdomain D_i , and select test input c from D_i at random
3. Add c into T

PB-ART

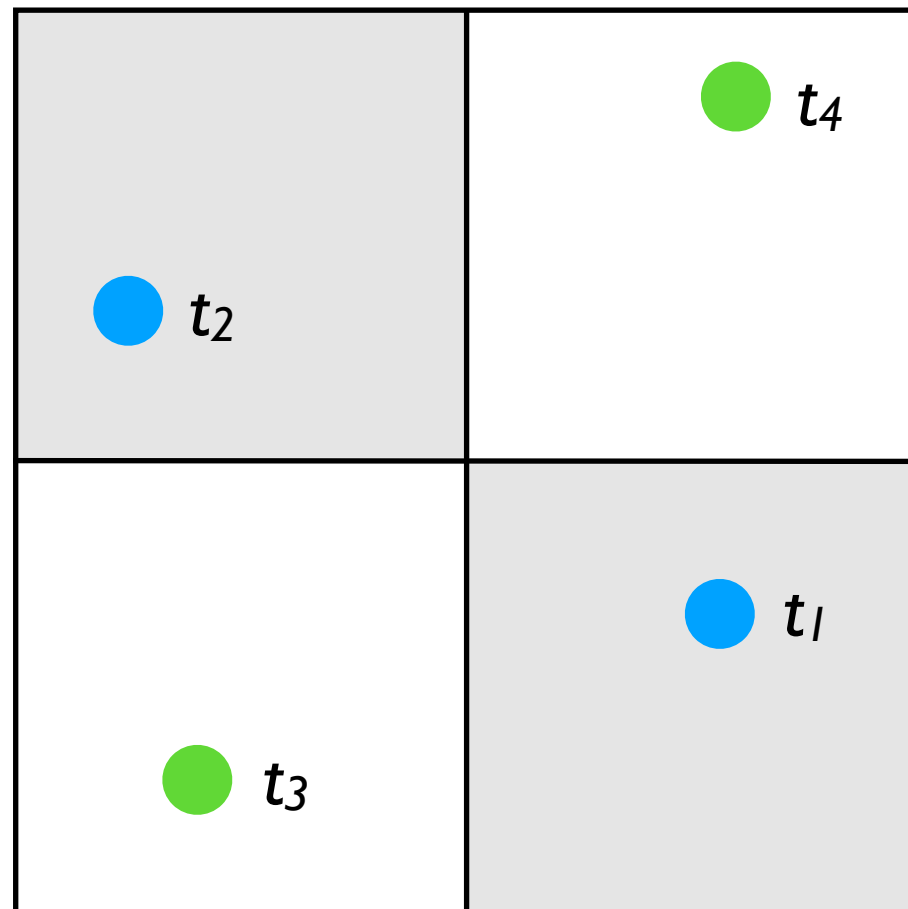
Partition Based-ART



假设使用 bisection partition

PB-ART

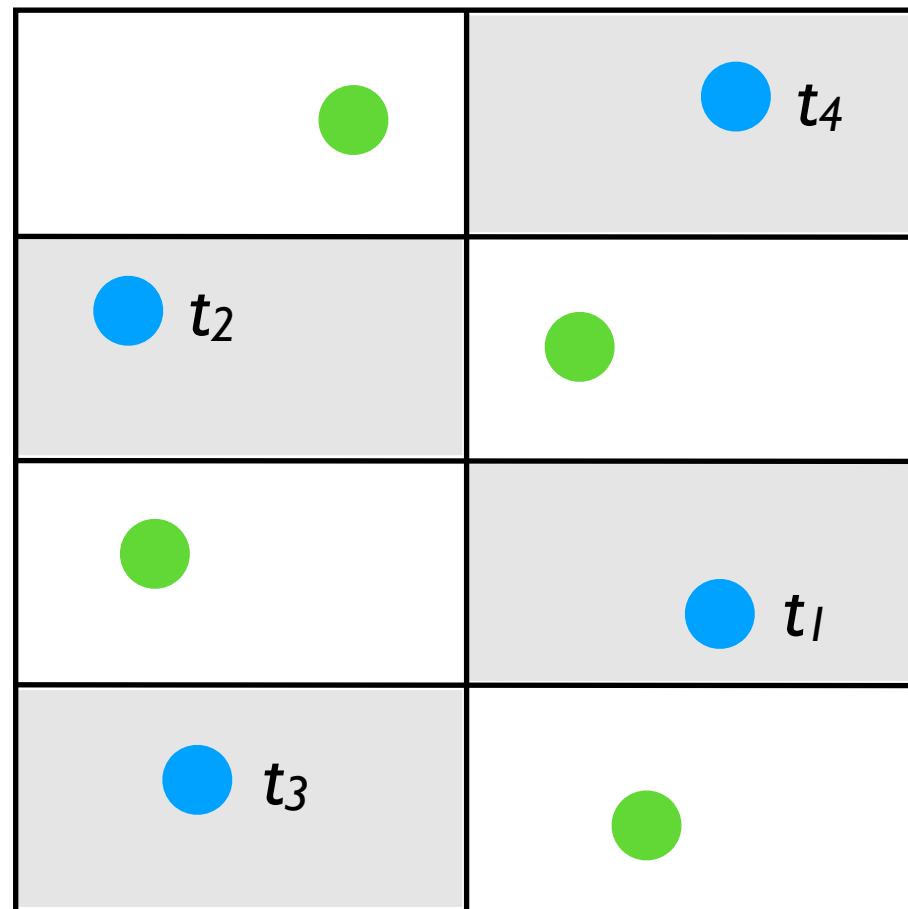
Partition Based-ART



假设使用 bisection partition

PB-ART

Partition Based-ART



假设使用 bisection partition

通过 Diversity 提升测试的有效性

Test Input Sampling (Selection) is all about *Diversity*

- 距离度量只是 Diversity 的一种表现形式
- 新的测试用例应该尽可能不同于已执行过的测试用例
(尽量避免已经测试过的区域)
 - 白盒测试: 不同的 Statements 和 Branches
 - 黑盒测试: 不同的 Partitions 和 Categories
 - 不同的测试方法
 - 不同的测试人员 (Crowdsourced Testing)
- 从输入空间的多样性到输出空间的多样性
 - 触发待测程序尽可能多样的行为