

# Advanced Software Testing and Debugging (CS598)

## Guided Unit Test Generation

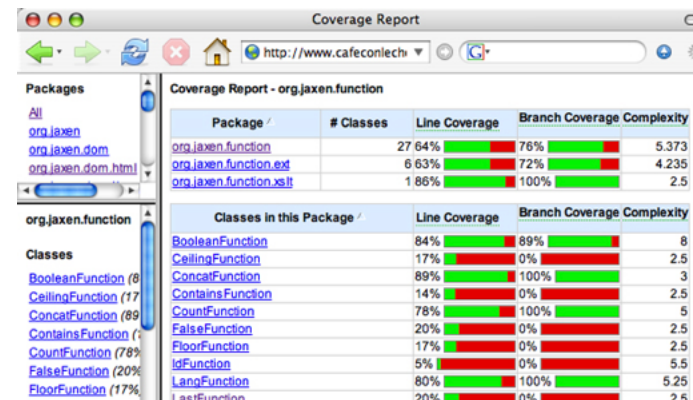
Fall 2020

Lingming Zhang



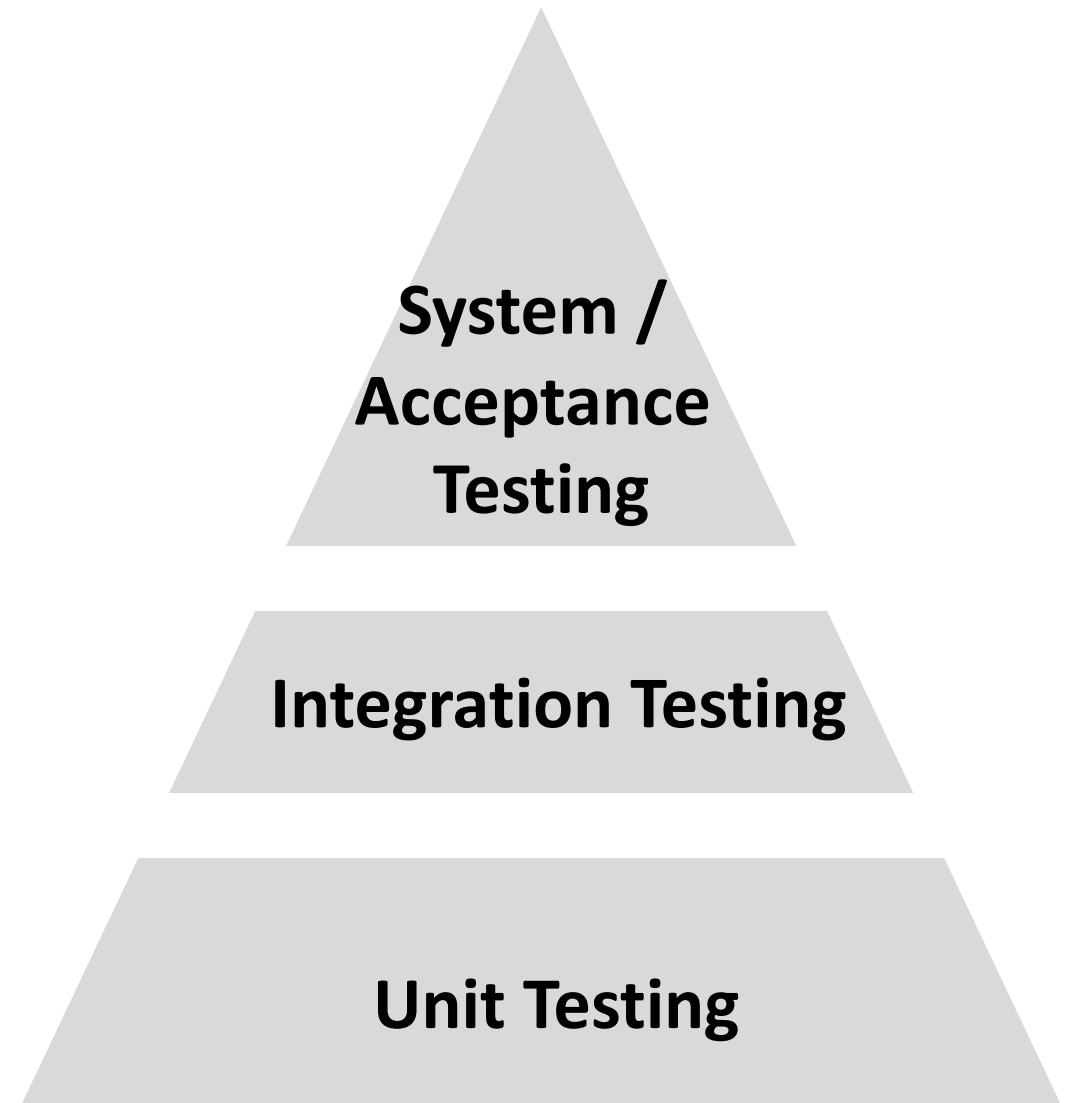
# Testing: basic concepts

- **Test case** (or, simply **test**): an execution of the software with a given test input, including:
  - Input values
  - Sometimes include execution steps
  - Expected outputs (**test oracle**)
- **Test suite**: a finite set of tests
  - Usually can be run together in sequence
- **Test adequacy**: a measurement to evaluate the test quality
  - Such as code coverage



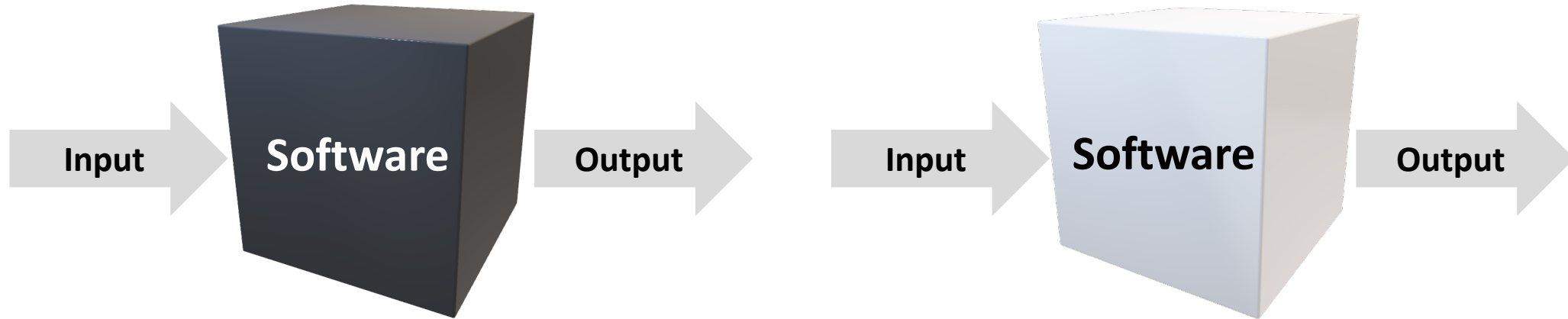
# Testing: levels

- Unit Testing
  - Test each single module in isolation
- Integration Testing
  - Test the interaction between modules
- System Testing
  - Test the system as a whole, by developers
- Acceptance Testing
  - Validate the system against user requirements, by customers with no formal test cases



# Types of test generation

- Black-box (functional) vs. white-box (structural) testing



- **Black-box test generation:** generates tests based on the functionality of the program
- **White-box test generation:** generates tests based on the source-code structure of the program

# White-box generation granularities

- Projects providing a number of public APIs for external use (e.g., JDK lib)
  - **Method-level test generation**: consider various method invocation sequences to expose possible faults

## Guided unit test generation (this class)

- Projects usually used as a whole (e.g., scientific computation software)
  - **Path-level generation**: consider all the possible execution paths to cover most program elements

## Symbolic execution (next class)

# This class

- Feedback-directed Random Test Generation (ICSE'07)
- Whole Test Suite Generation (TSE'12)

# Problem: unit test generation

## Program under test:

```
public class Math{  
    public static int sum(int a, int b){  
        return a+b;  
    }  
    ...  
}
```

## Example JUnit test:

```
public class MathTest{  
    @Test  
    public void testSum (){  
        int a=1;  
        int b=1;  
        int c=Math.sum(a, b);  
        assertEquals(2,c);  
    }  
    ...  
}
```

Input values

Execution steps

Test oracle

## Is this an important problem?



{ 84,377 lines of **source code**  
86,924 lines of **unit-test code**

# How to perform random white-box test generation?

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

Program under test

Generation

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

Generated test *t1*

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

Generated test *t2*

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

Generated test *t3*

- Need to generate a random sequence of invocations, where each requires
  - A random method
  - Some random parameters
  - A random receiver object
    - Not required for static methods

...



# Random method-sequence generation: limitations

- Does not have test oracles
  - E.g., an idea test oracle for the test below: **assertEquals(1, s.size())**
- Cannot generate complex tests
  - E.g., the parameters of some method invocations can be generated by other method invocations
- Can have many redundant&illegal tests

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

A random test

# Random method-sequence generation: redundant&illegal tests

## 1. Useful test:

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

## 2. Redundant test:

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

Should not output

## 3. Useful test:

```
Date d = new Date(2006, 2, 14);
```

## 4. Illegal test:

```
Date d = new Date(2006, 2, 14);  
d.setMonth(-1); // pre: argument >= 0
```

Should not output

## 5. Illegal test:

```
Date d = new Date(2006, 2, 14);  
d.setMonth(-1); // pre: argument >= 0  
d.setDay(5);
```

Should not even generate

# Randoop: feedback-directed (adaptive) random test generation

- Use code contracts as test oracles
- Build test inputs incrementally
  - New test inputs extend previous ones
  - In this context, a test input is a method sequence
- As soon as a test is created, use its execution results to guide generation
  - away from redundant or illegal method sequences
  - towards sequences that create new object states

# Randoop input/output

- **Input:**

- Classes under test
- Time limit
- Set of contracts
  - Method contracts (e.g. “o.hashCode() throws no exception”)
  - Object invariants (e.g. “o.equals(o) == true”)

- **Output:** contract-violating test cases

```
HashMap h = new HashMap();  
Collection c = h.values();  
Object[] a = c.toArray();  
LinkedList l = new LinkedList();  
l.addFirst(a);  
TreeSet t = new TreeSet(l);  
Set u = Collections.unmodifiableSet(t);  
assertTrue(u.equals(u));
```



fails on Sun's JDK 1.5/1.6  
when executed

# Randoop: algorithm

- Seed value pool for primitive types
  - pool = { 0, 1, true, false, "hi", null ... }
- Do until time limit expires:
  - Create a new sequence
    - Randomly pick a method call  $\mathbf{m}(T_1 \dots T_k) / T_{ret}$
    - For each input parameter of type  $T_i$ , randomly pick a sequence  $S_i$  from the value pool that constructs an object  $\mathbf{v}_i$  of type  $T_i$
    - Create new sequence  $S_{new} = S_1; \dots ; S_k ; T_{ret} \mathbf{v}_{new} = \mathbf{m}(\mathbf{v}_1 \dots \mathbf{v}_k)$ ;
    - if  $S_{new}$  was previously created (lexically), go to first step
  - Classify the new sequence  $S_{new}$ 
    - May discard, output as test case, or add to pool

- - - -> Method
- - - -> Parameter
- - - -> Receiver object

# Randoop: example

## Program under test:

```
public class A{
    public A() {...}
    public B m1(A a1) {...}
}
public class B{
    public B(int i) {...}
    public void m2(B b, A a) {...}
}
```

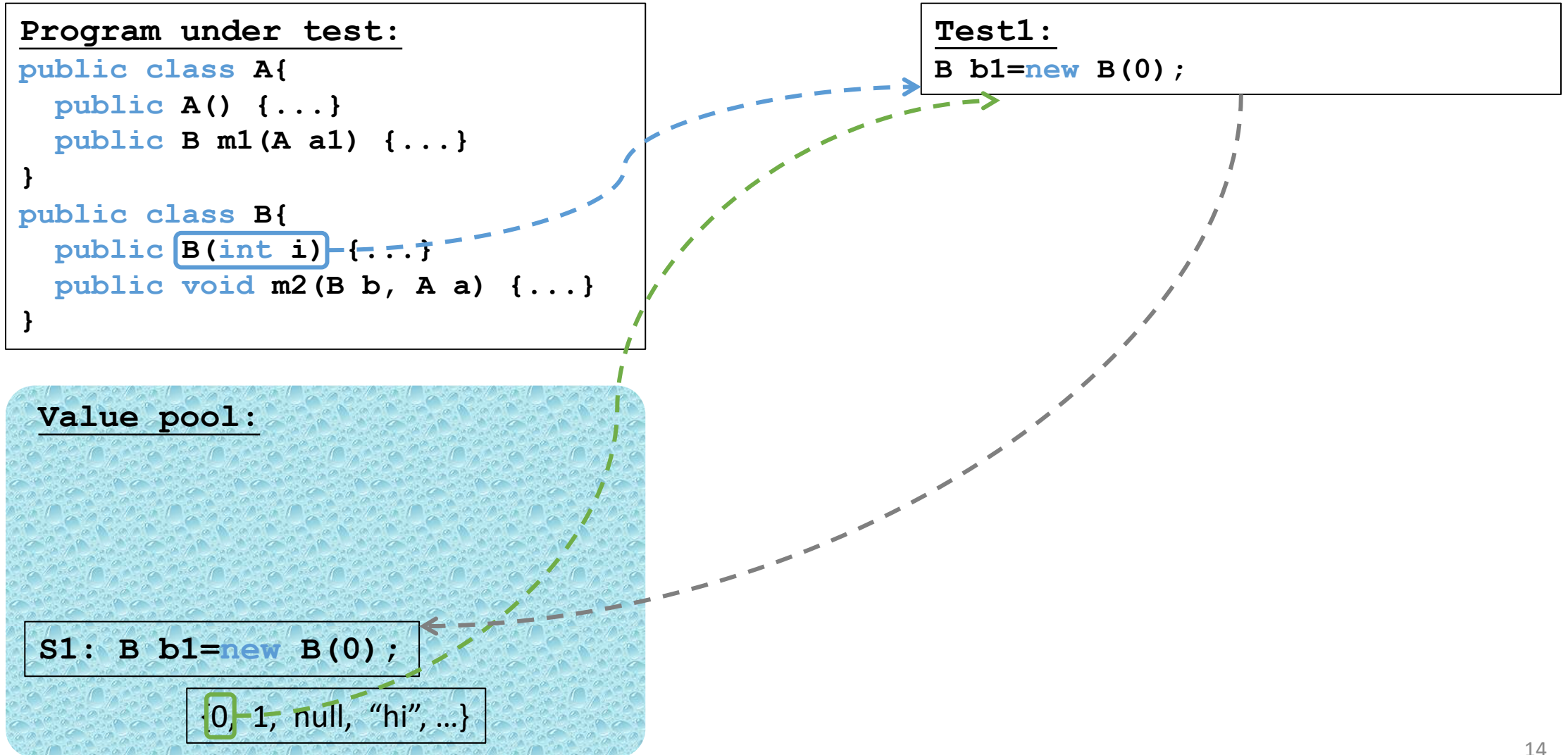
## Test1:

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

## Value pool:

```
S1: B b1=new B(0);
```

```
{0, 1, null, "hi", ...}
```



- - - → Method
- - - → Parameter
- - - → Receiver object

# Randoop: example

## Program under test:

```
public class A{
    public A() {...}
    public B m1(A a1) {...}
}
public class B{
    public B(int i) {...}
    public void m2(B b, A a) {...}
}
```

## Test1:

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

## Test2:

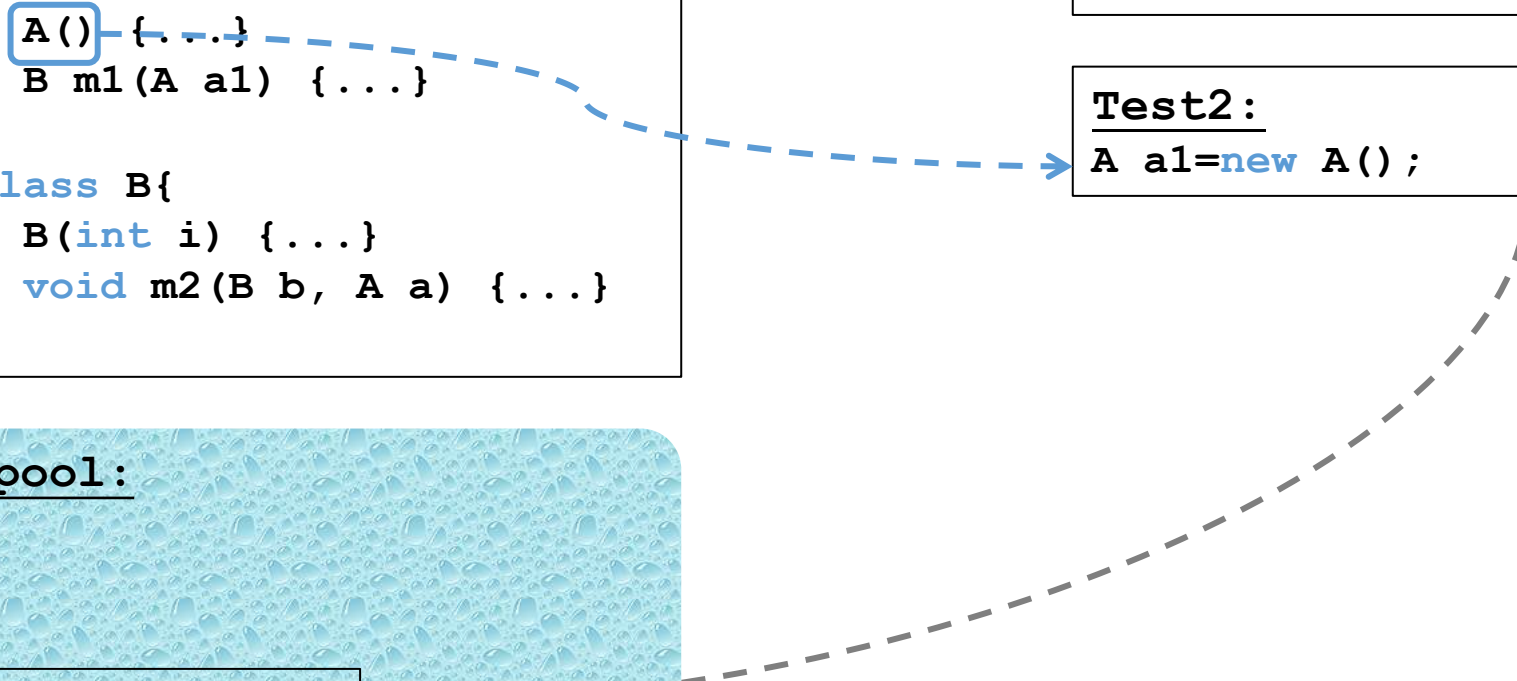
```
A a1=new A();
```

## Value pool:

```
S2: A a1=new A();
```

```
S1: B b1=new B(0);
```

```
{0, 1, null, "hi", ...}
```



- - - -> Method
- - - -> Parameter
- - - -> Receiver object

# Randoop: example

## Program under test:

```
public class A{
    public A() {...}
    public B m1(A a1) {...}
}
public class B{
    public B(int i) {...}
    public void m2(B b, A a) {...}
}
```

## Test1:

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

## Test2:

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

## Test3:

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

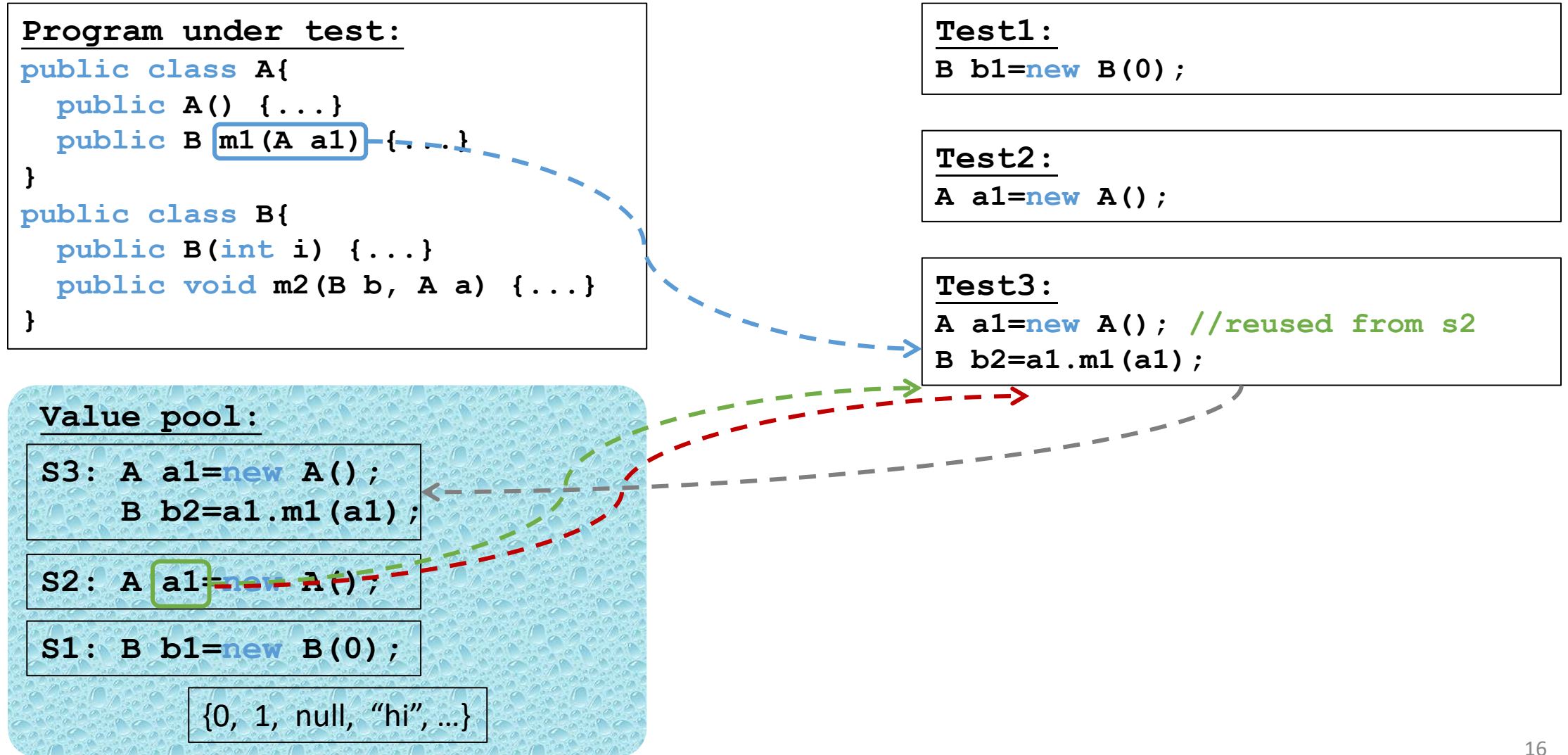
## Value pool:

```
S3: A a1=new A();
    B b2=a1.m1(a1);
```

```
S2: A a1=new A();
```

```
S1: B b1=new B(0);
```

```
{0, 1, null, "hi", ...}
```





- - - → Method
- - - → Parameter
- - - → Receiver object

# Randoop: example

## Program under test:

```
public class A{
    public A() {...}
    public B m1(A a1) {...}
}
public class B{
    public B(int i) {...}
    public void m2(B b, A a) {...}
}
```

## Test1:

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

## Test2:

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

## Test3:

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

## Test4:

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

...

## Value pool:

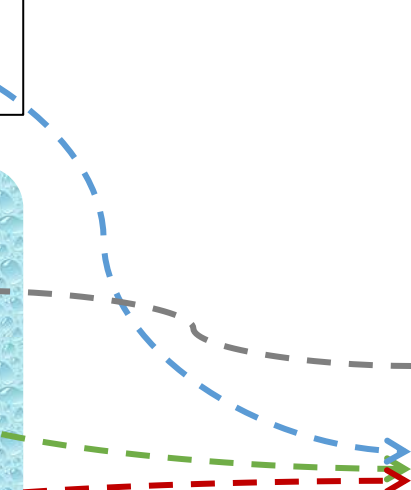
S3: A a1=new A();  
B b2=a1.m1(a1);

S4: ...

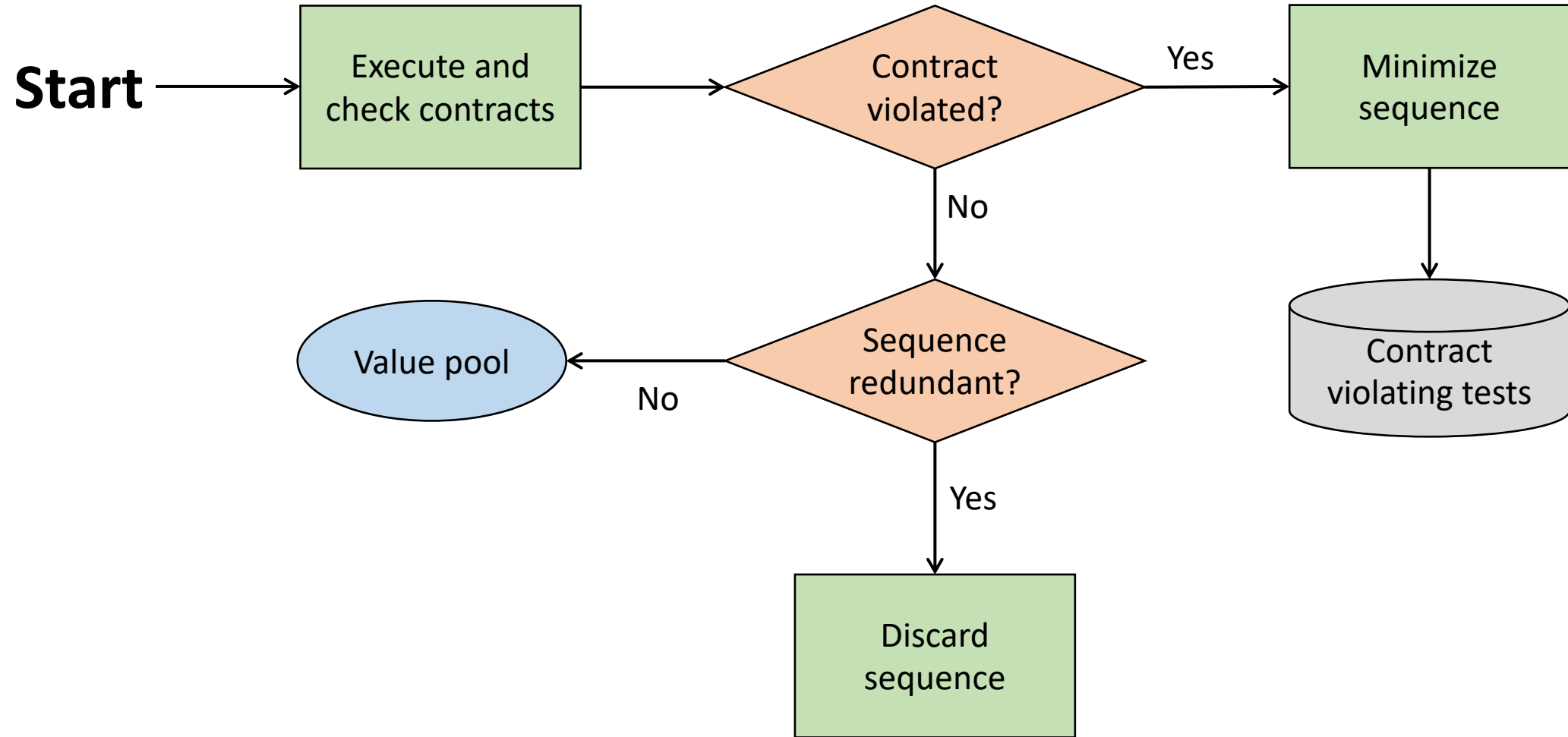
S2: A a1=new A();

S1: B b1=new B(0);

{0, 1, null, "hi", ...}



# Classifying a sequence



# Redundant sequences

- During generation, maintain a set of all objects created
- A sequence is redundant if all the objects created during its execution are members of the above set (using *equals* to compare)
- Could also use more sophisticated state equivalence methods
  - E.g. heap canonicalization

# Tool support

- **Input:**
  - An assembly (for .NET) or a list of classes (for Java)
  - Generation time limit
  - Optional: a set of contracts to augment default contracts
- **Output:** a test suite (JUnit or NUnit) containing
  - Contract-violating test cases
  - Normal-behavior test cases

# Randoop outputs oracles

- Oracle for contract-violating tests:

```
Object o = new Object();  
LinkedList l = new LinkedList();  
l.addFirst(o);  
TreeSet t = new TreeSet(l);  
Set u = Collections.unmodifiableSet(t);  
assertTrue(u.equals(u)); //expected to fail
```

Find **current** bugs

- Oracle for normal-behavior tests (regression tests):

```
Object o = new Object();  
LinkedList l = new LinkedList();  
l.addFirst(o);  
l.add(o);  
assertEquals(2, l.size()); //expected to pass  
assertEquals(false, l.isEmpty()); //expected to pass
```

Find **future** bugs

# Some Randoop options

- Avoid use of null

## Statically:

```
Object o = new Object();  
LinkedList l = new LinkedList();  
l.add(null);
```

## Dynamically:

```
Object o = returnNull();  
LinkedList l = new LinkedList();  
l.add(o);
```

- Bias random selection
  - Favor shorter sequences
  - Favor methods that have been less covered
  - Use constants mined from source code
- Source code available:
  - <https://randoop.github.io/randoop/>

# Code coverage by Randoop

<b>Data structure programs</b>	<b>Time (s)</b>	<b>Branch cov.</b>
Bounded stack (30 LOC)	1	100%
Unbounded stack (59 LOC)	1	100%
BS Tree (91 LOC)	1	96%
Binomial heap (309 LOC)	1	84%
Linked list (253 LOC)	1	100%
Tree map (370 LOC)	1	81%
Heap array (71 LOC)	1	100%

# Bug detection by Randoop: subjects

Subjects	LOC	Classes
JDK (2 libraries) (java.util, javax.xml)	53K	272
Apache commons (6 libraries) (logging, primitives, chain, jelly, math, collections)	114K	974
.Net libraries (6 libraries)	615K	3455



# Bug detection by Randoop: methodology

- Ran Randoop on each library
  - Used default time limit (2 minutes)
- Contracts:
  - **`o.equals(o)==true`**
  - **`o.equals(o)`** throws no exception
  - **`o.hashCode()`** throws no exception
  - **`o.toString()`** throw no exception
  - No null inputs and:
    - Java: No NPEs
    - .NET: No NPEs, out-of-bounds, of illegal state exceptions

# Bug detection by Randoop: subjects

<b>Subjects</b>	<b>Failed tests</b>	<b>Unique failed tests</b>	<b>Error-revealing tests</b>	<b>Distinct errors</b>
JDK	613	32	29	8
Apache commons	3,044	187	29	6
.Net framework	543	205	196	196
Total	4,200	424	254	210

# Errors found: examples

- JDK Collections classes have 4 methods that create objects violating **`o.equals(o)`** contract
- Javax.xml creates objects that cause **`hashCode`** and **`toString`** to crash, even though objects are well-formed XML constructs
- Apache libraries have constructors that leave fields unset, leading to NPE on calls of **`equals`**, **`hashCode`** and **`toString`** (this only counts as one bug)
- .Net framework has at least 175 methods that throw an exception forbidden by the library specification (NPE, out-of-bounds, of illegal state exception)
- .Net framework has 8 methods that violate **`o.equals(o)`**
- .Net framework loops forever on a legal but unexpected input

# Has Randoop been compared to existing solutions?

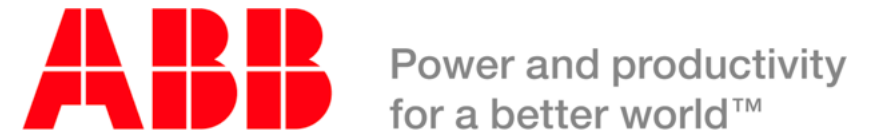
- Systematic testing:
  - Java PathFinder (JPF)
  - jCUTE
- Undirected Random testing:
  - Randoop-feedback
  - JCrasher

# Regression testing scenario

- Randoop can create regression oracles
- Generated test cases using JDK 1.5
  - Randoop generated 41K regression test cases
- Ran resulting test cases on
  - JDK 1.6 Beta
    - 25 test cases failed
  - Sun's implementation of the JDK
    - 73 test cases failed
  - Failing test cases pointed to 12 distinct errors
  - These errors were not found by the extensive compliance test suite that Sun provides to JDK developers

```
Object o = new Object();  
LinkedList l = new LinkedList();  
l.addFirst(o);  
l.add(o);  
assertEquals(2, l.size()); //expected to pass  
assertEquals(false, l.isEmpty()); //expected to pass
```

# Randoop: applications



# Discussion

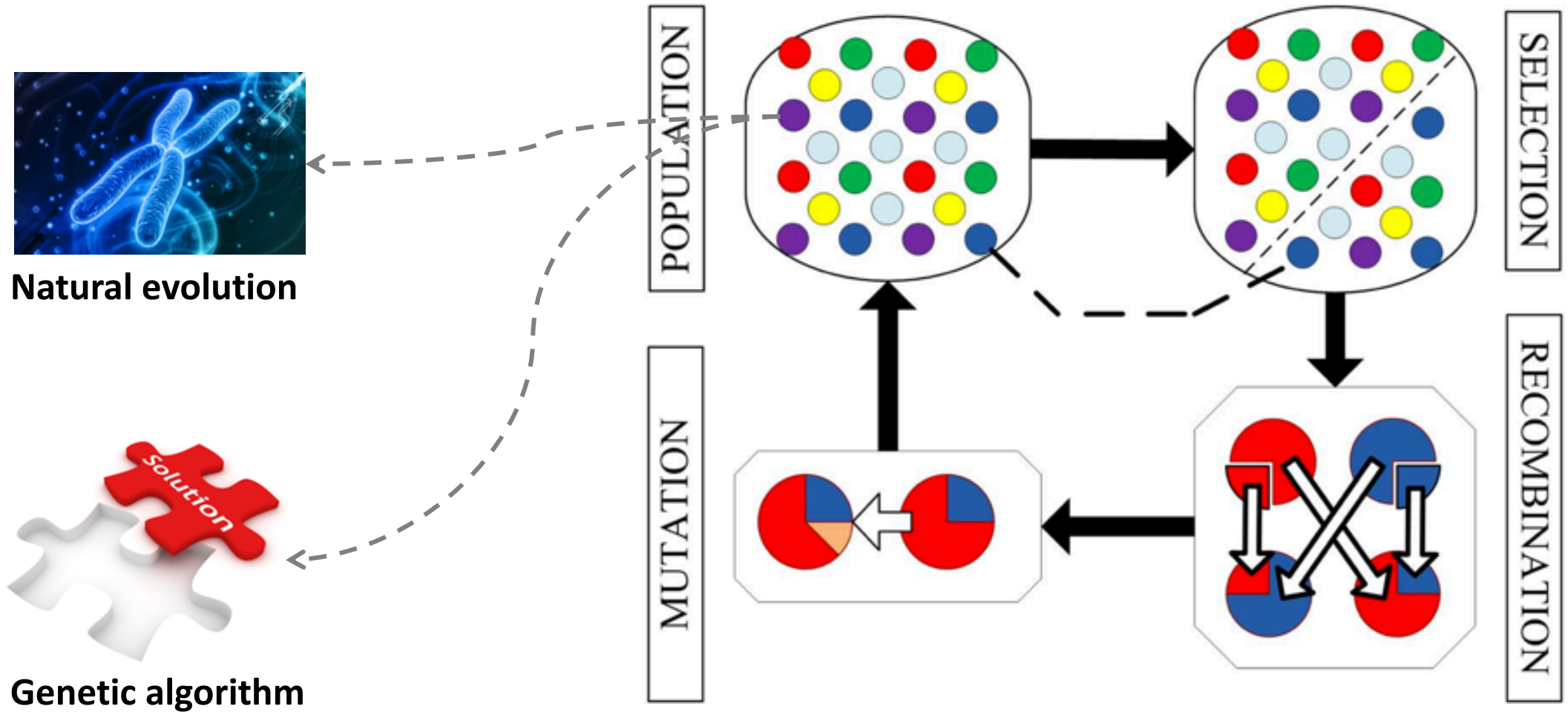
- Strengths
- Limitations
- Future work

# This class

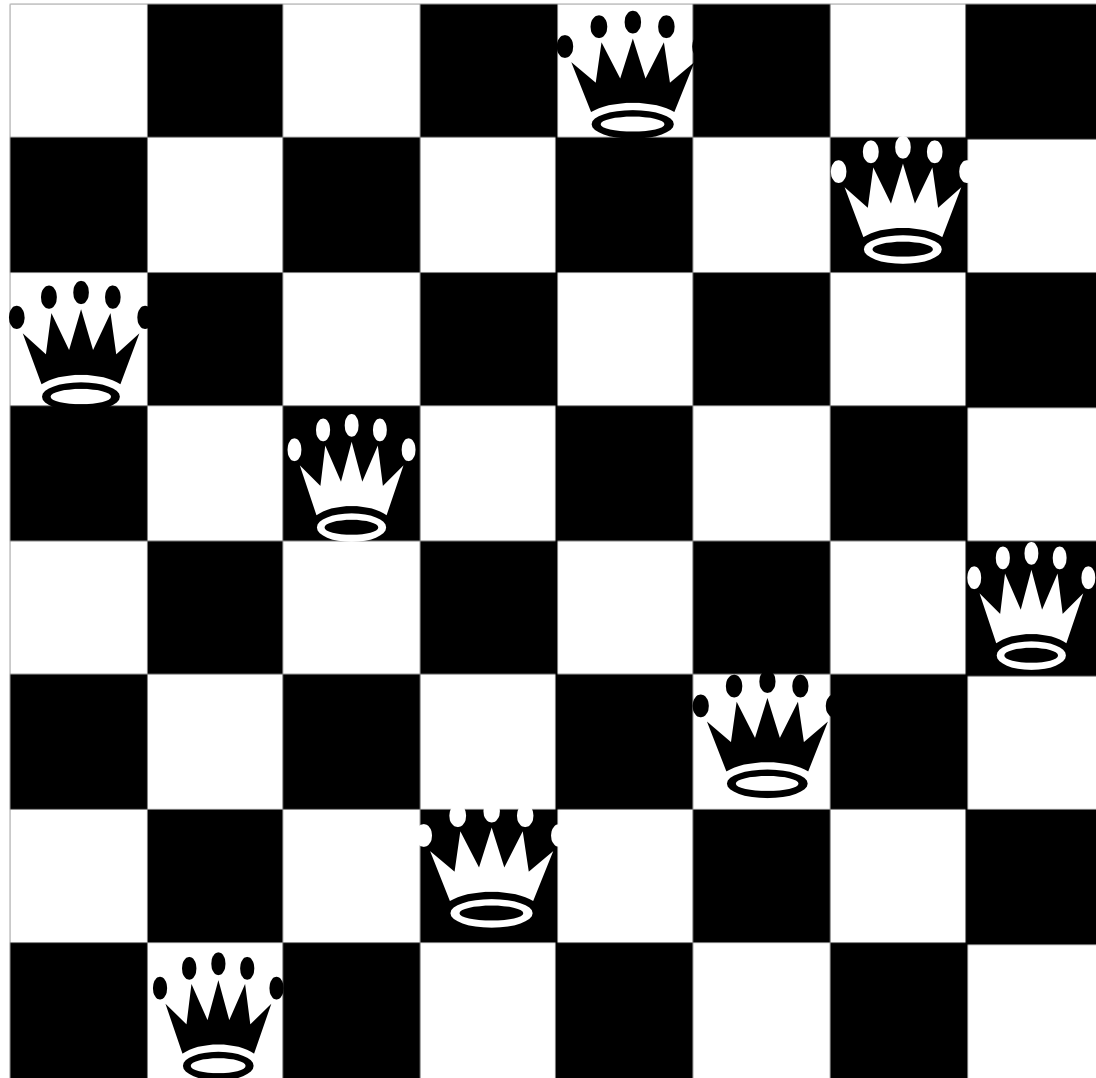
- Feedback-directed Random Test Generation (ICSE'07)
- Whole Test Suite Generation (TSE'12)



# Genetic algorithm

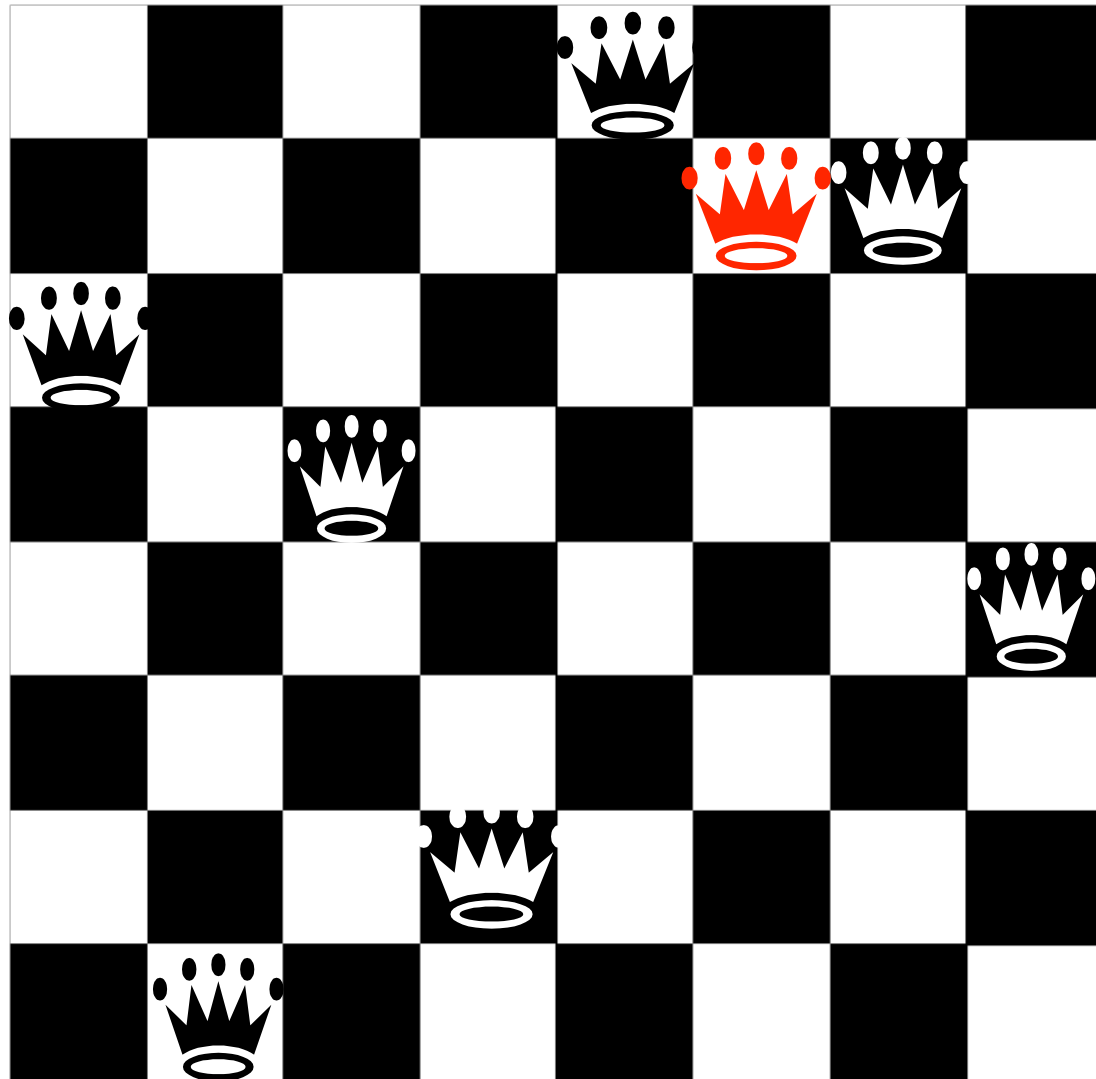


# The eight queens problem



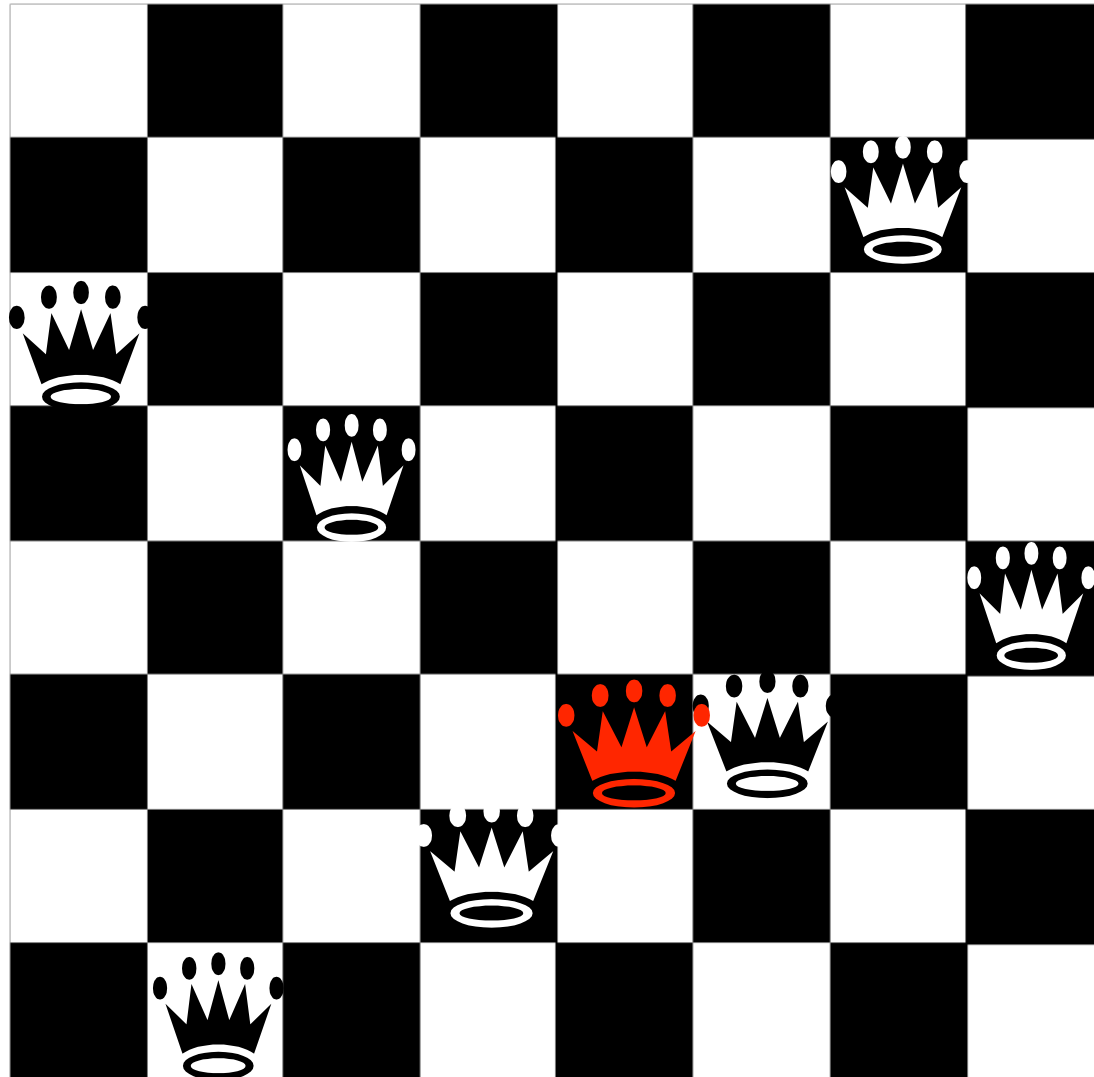
**Perfect!**

# The eight queens problem



**2 attacks!**

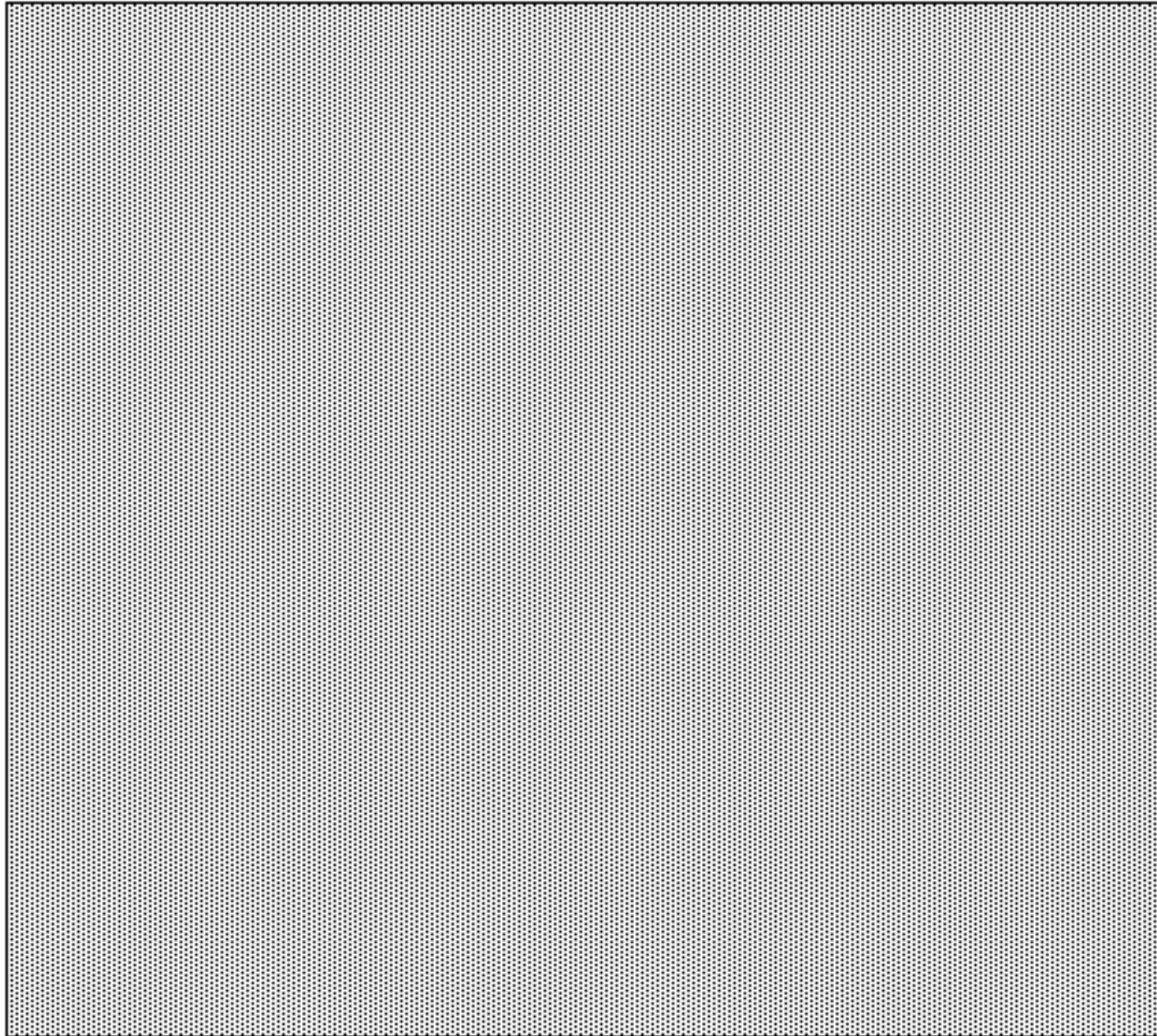
# The eight queens problem



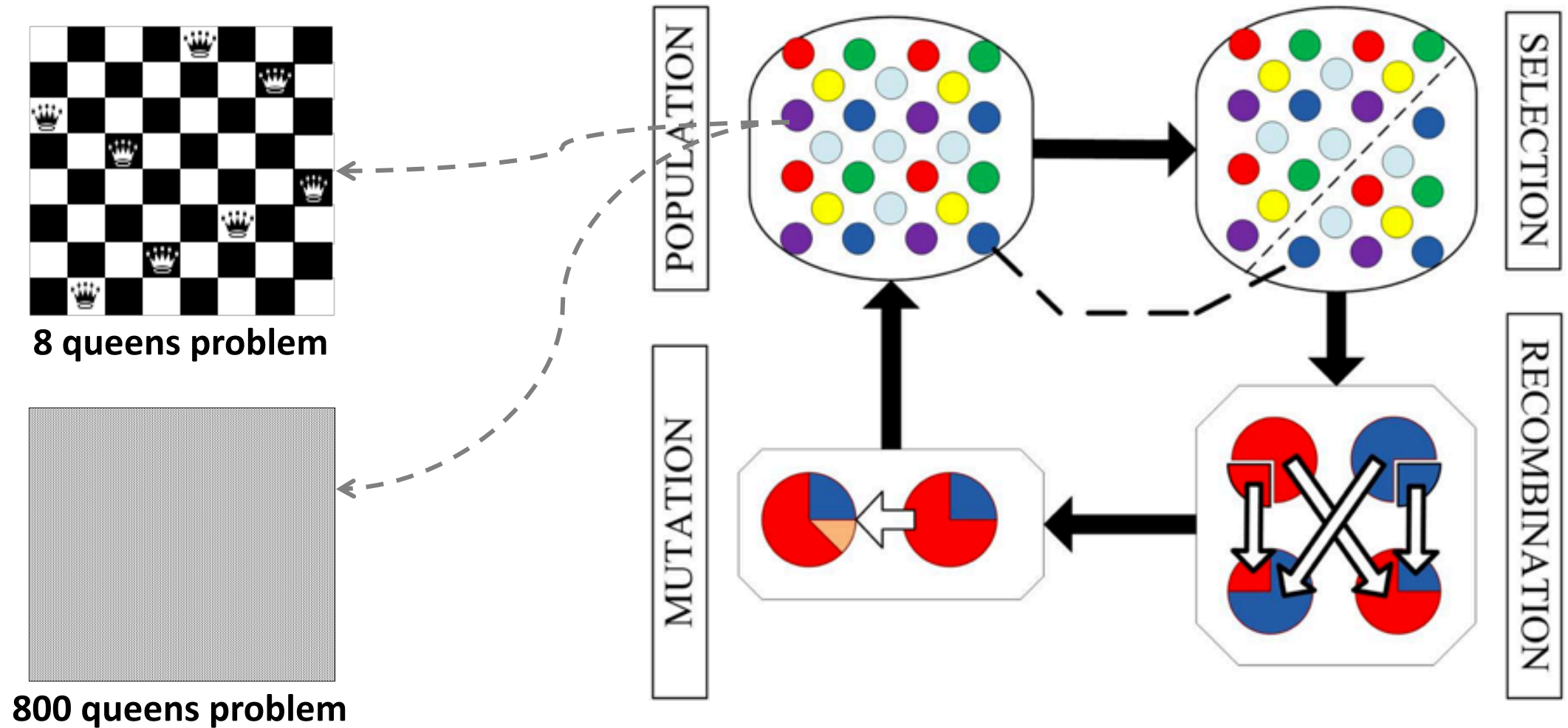
**3 attacks!**

Easily solved via recursion or  
dynamic programming!

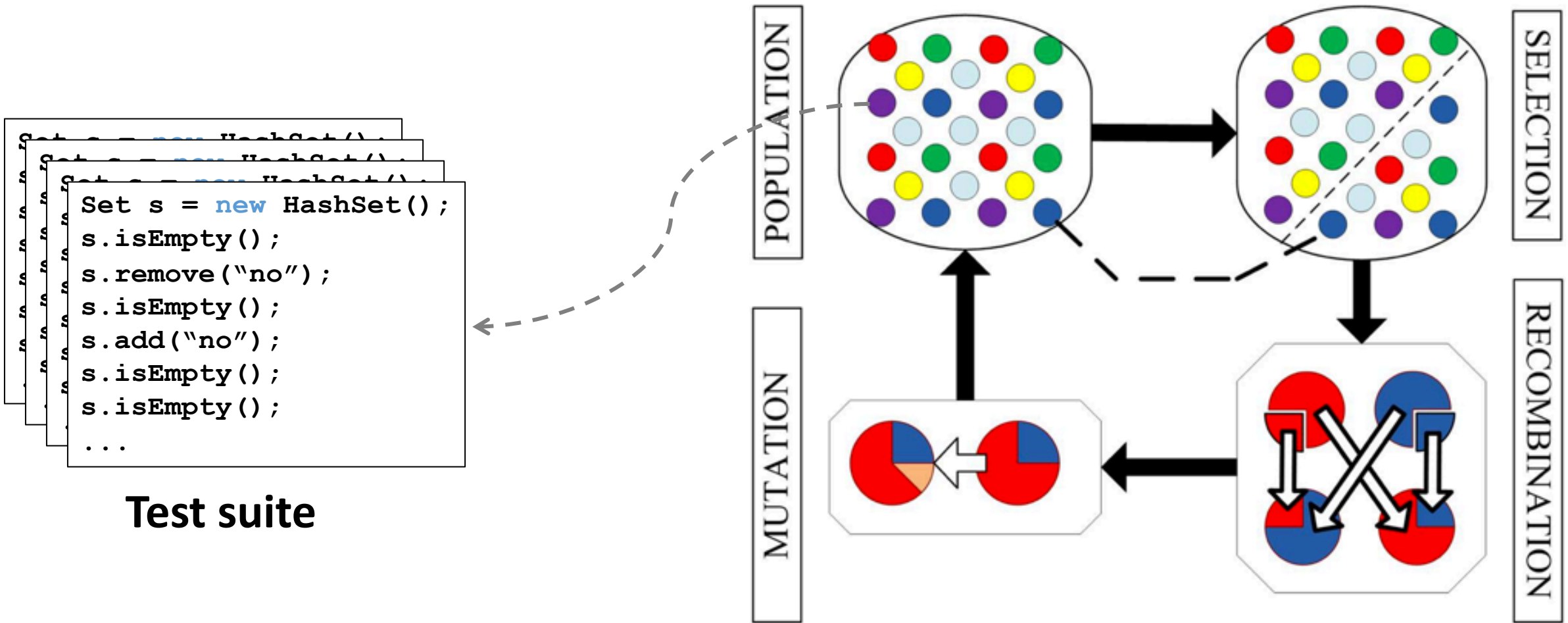
# How about 800 queens problem?!



# Genetic algorithm: 8/800 queens problem



# Genetic algorithm: test generation (aka search-based test generation)



# Crossover and mutation

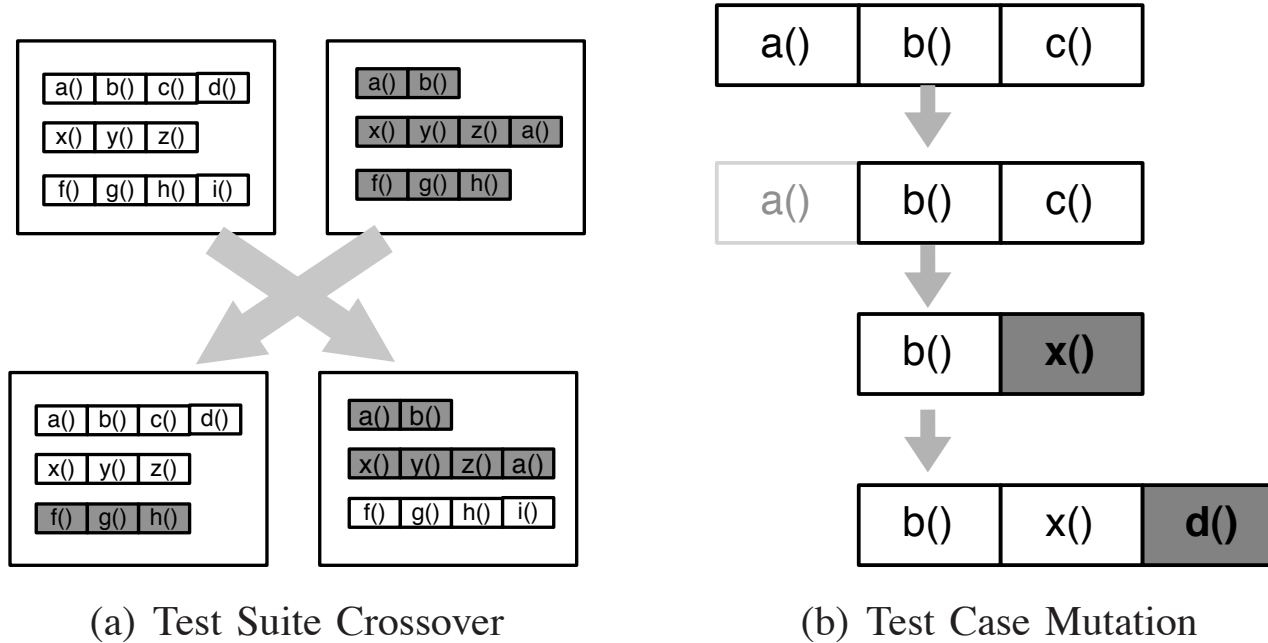


Fig. 3. Crossover and mutation are the basic operators for the search using a GA. Crossover is applied at test suite level; mutation is applied to test cases and test suites.

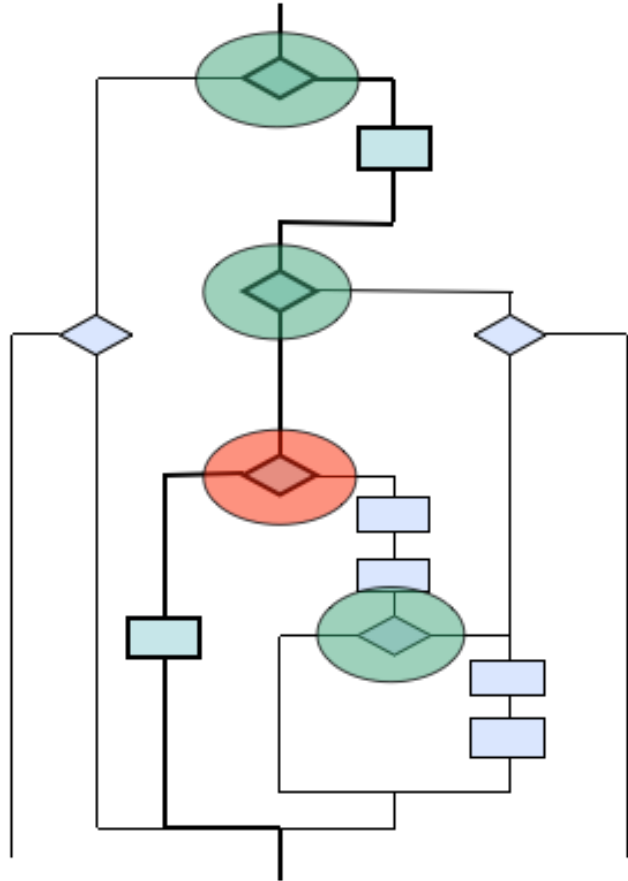


# Fitness function and selection

All methods  
in program

Methods  
covered by  $T$

All branches  
in program



- $fitness(T) = |M| - |M_T| + \sum_{b \in B} dist(b, T)$

- $dist(b, T) = \begin{cases} 0 & \text{If the branch is covered} \\ d(b, T) & \text{If the predicate is executed twice} \\ 1 & \text{Otherwise} \end{cases}$

**Branch distance,  $d(b, T)$ , describes how “close” branch  $b$  is to being covered**

$$\text{if } A = B \rightarrow d(b, T) = |A - B|$$

# Discussion

- Strengths
- Limitations
- Future work

Thanks and stay safe!