CO3091 - Computational Intelligence and Software Engineering

Lecture 14

Image from: http://www.teluguone.com/teluguoneUserFiles/it%20joke.png

Evolutionary Software Testing

Leandro L. Minku

Announcements

- Problem class tomorrow.
- Extra surgery.

Coursework

- Objective: to determine the impact of the parameter values on the fitness of the final solution.
- Part 1: perform runs.
- Part 2: perform statistical tests — these only enable you to tell whether or not there is a significant impact.
- Part 3: analyse the results further — what is the impact?

Overview

- Software testing and the importance of intelligent test automation.
- Formulation of test suite generation for the purpose of finding crashes as an optimisation problem.
- NSGA-II design to solve this problem.

Software Testing

- Software testing is an essential component for software development success.
- Software testing is one of the most expensive tasks in the software development process.

```java
protected double determineUnnormSumCurrDedicationEmployee(int e, Vector<Integer> curIndependentTasks, PSPPhenotype phen) {
	double sum = 0d;
	for (int i=0; i<curIndependentTasks.size(); ++i) {
		int t = curIndependentTasks.get(i);
		sum += phen.getEmployeeTaskDedication(e, t);
	}
	return sum;
}
```

method under test

unit test

```java
public void testDetermineUnnormSumDedicationEmployee() {
	PSPPhenotype phen = new PSPPhenotype(4, 5);
	for (int e=0; e<4; ++e)
		for (int t=0; t<5; ++t)
			phen.setEmployeeTaskDedication(e, t, e+t);

	Vector<Integer> curIndpTasks = new Vector<Integer>(3);
	curIndpTasks.add(0);
	curIndpTasks.add(3);
	curIndpTasks.add(1);

	assertEquals(7d, eval.determineUnnormSumCurrDedicationEmployee(1, curIndpTasks, phen));

	for (int e=0; e<4; ++e)
		for (int t=0; t<5; ++t)
			phen.setEmployeeTaskDedication(e, t, (e+t)/100d);

	assertEquals(0.07d, eval.determineUnnormSumCurrDedicationEmployee(1, curIndpTasks, phen));

	PSPPhenotype phen2 = new PSPPhenotype(1,2);
	for (int e=0; e<1; ++e)
		for (int t=0; t<2; ++t)
			phen2.setEmployeeTaskDedication(e, t, 1);

	curIndpTasks = eval.determineCurIndependentTasks(inst.tpg.inDegreeClone());
	assertEquals(1d, eval.determineUnnormSumCurrDedicationEmployee(0, curIndpTasks, phen2));
}
```

Software Test Suite Generation

- Test suite: set of test cases, each consisting of a sequence of inputs and expected outputs from the program.
- Challenging for large and complex software.
- A good test suite should exercise the code well and be fast to run.

Evolutionary Software Test Suite Generation

- Evolutionary algorithms have been used to aid the generation of test suites with the objectives of:
  - Maximising coverage and minimising the length of test cases (fast to run).
- They can generate input sequences for test cases.
- Identification of buggy behaviour requires expected outputs to be defined by humans.
- Helpful specially for life-critical applications.

Image from: https://cnet2.cbsistatic.com/img/vNg3GpX08rMV8J1ZxWIQ-2ligg8=/fit-in/970x0/2013/08/27/6d22dd41-6de7-11e3-913e-14feb5ca9861/Nissan-leaf-autonomous_1.jpg

Image from: https://img.purch.com/h/1000/aHR0cDovL3d3dy5zcGFjZS5jb20vaW1hZ2VzL2kvMDAwLzAxOC8zNjQvb3JpZ2luYWwvc2xzLXJvY2tldC1hcnQuanBn

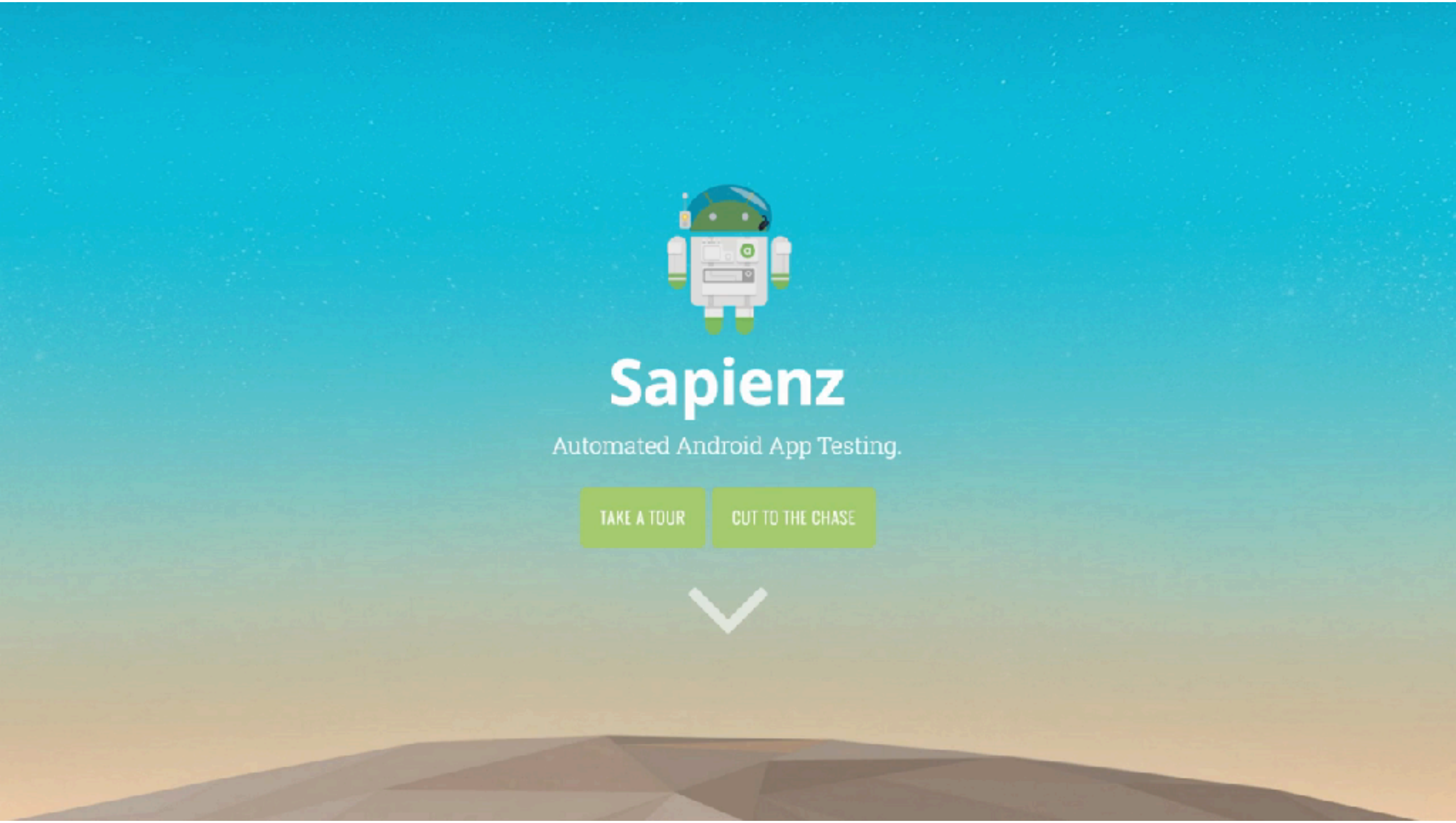
Evolutionary Software Testing for the Purpose of Searching for Crashes

- Software crashes can greatly affect users' satisfaction and trust:
  - Serious software failure.
  - Software stops working properly and aborts unexpectedly.
  - Frequently caused by bugs.
- Evolutionary algorithms can be particularly helpful to automate test suit generation to search for crashes.
  - Input sequences that reveal crashes could be considered as bug-revealing.
  - Humans don't need to specify the desired output for a given input sequence — expect to see "no crash".
  - This makes evolutionary software testing useful for a very wide range of applications.

Why Not Generating Test Suites Randomly or Systematically?

- Completely at random:
    - Takes a long time to find crashes.
        - >15,000 test inputs between crashes.
    - Very large test cases.
        - Time consuming to run.
        - Difficult to debug.
- Systematically generate all possible combinations of inputs:
    - Similar problems as above.
    - Too many different combinations of inputs.
- Evolutionary algorithms:
    - Around 100 to 150 test inputs tested between crashes.
    - Find much more crashes than the approaches above within a limited amount of time.

Sapienz — Tool based on NSGA-II to Search for Crashes



Video from: https://youtu.be/j3eV8NiWLg4

Sapienz — Historical Facts

- Sapienz was developed by researchers from UCL in 2016.
  - Tested top 1000 most popular Google Play apps and found 558 unique previously unknown crashes.
- They created a spinout company called MaJiCkE.
- Facebook bought MaJiCkE.

http://www.engineering.ucl.ac.uk/news/bug-finding-majicke-finds-home-facebook/

https://arstechnica.co.uk/information-technology/2017/08/facebook-dynamic-analysis-software-sapienz/

Problem Formulation

- Design variable: list with a given number of input sequences.
  - Design variable can be seen as a test suite.
  - Each input sequence is a test case with an expected “output” of “no crash”.
  - Examples of inputs for Android: Touch, Motion, Rotation, Track-ball, PinchZoom, Flip, Nav, MajorNav, AppSwitch, SysOp, enter text, or clicks on widgets.

- Objectives:
  - Maximise coverage.
  - Minimise length of test cases.
  - Maximise number of crashes found.

- Constraints:
  - N/A

NSGA-II Design Behind Sapienz

Problem Formulation

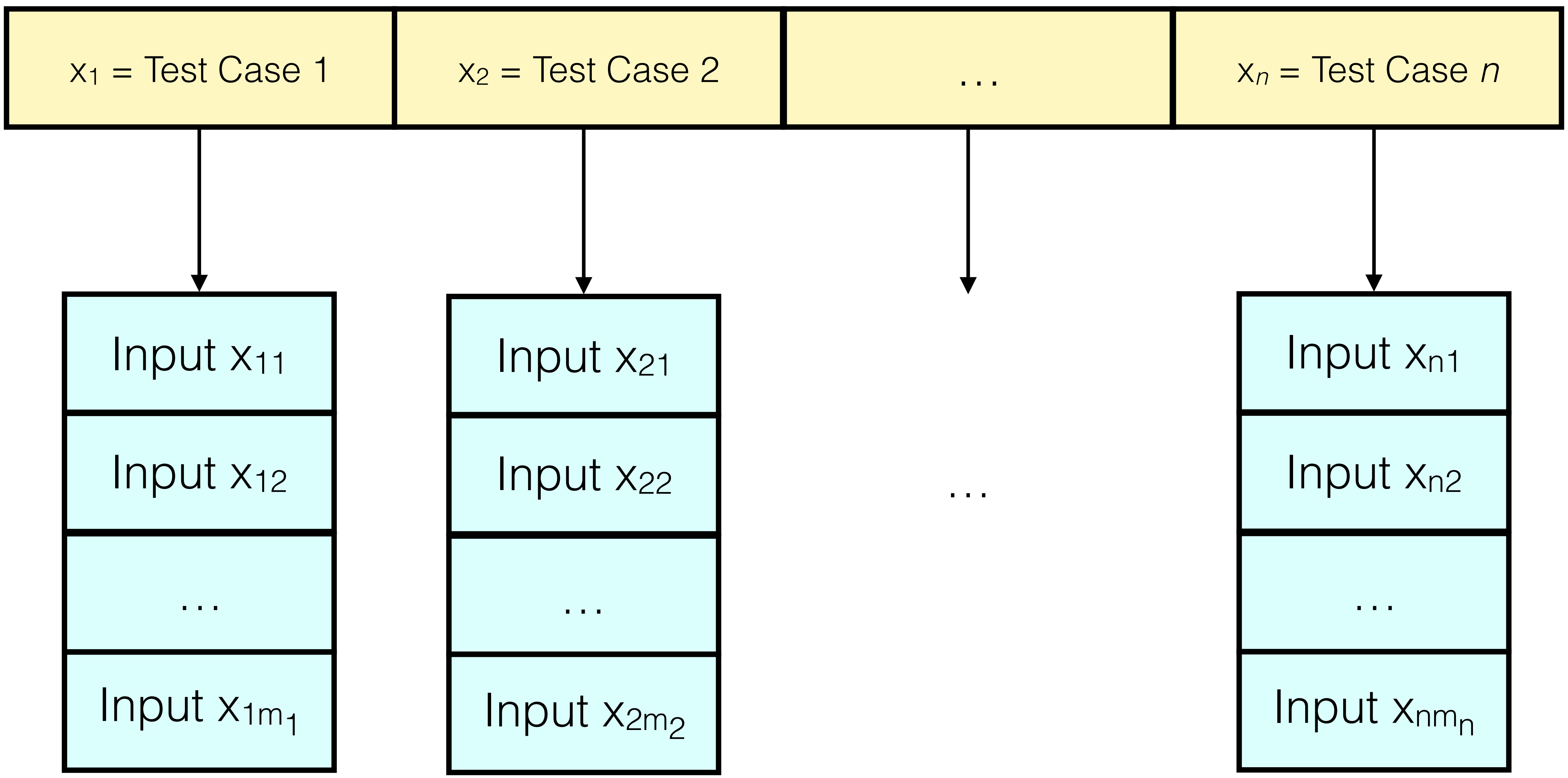
- **Design variable: list with a given number of input sequences.**
  - Design variable can be seen as a test suite.
  - Each input sequence is a test case with an expected “output” of “no crash”.
  - Examples of inputs for Android: Touch, Motion, Rotation, Track-ball, PinchZoom, Flip, Nav, MajorNav, AppSwitch, SysOp, enter text clicks on widgets.

- Objectives:
  - Maximise coverage.
  - Minimise length of test cases.
  - Maximise number of crashes found.

- Constraints:
  - N/A

Representation

Individual **x** (test suite):



The value of n is fixed, but m_i ($1 \leq i \leq n$) is variable.

Representation — Inputs

Each input can be an atomic event or a motif event:

Atomic Event

An atomic event can be one of:

- Touch,
- Motion,
- Rotation,
- Track-ball,
- PinchZoom,
- Flip,
- Nav,
- MajorNav,
- AppSwitch, or
- SysOp

Motif Event

A motif event is a compound of several inputs as follows:

- inputs for entering strings into each text field under the corresponding view; and
- an attempt to exercise each clickable widget to transfer to the next view.

Initialisation

- Create individuals containing *n* test cases each, where *n* is a pre-defined value.
- Each test case contains a number of inputs between [1,Max], where Max is a pre-defined maximum number of inputs for initialisation purposes.
- Each input is randomly picked as an atomic event or a motif event.
- Parameters of the input (e.g., coordinates of touch) can be generated randomly.

Entering Strings

- Apps such as Facebook use lots of human-generated content.
- In order to generate realistic strings, enter strings statically defined strings from within the code.
- Alternatively, enter dummy strings, e.g., “0”.

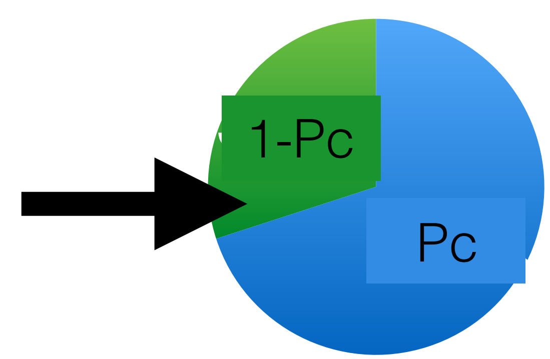
Higher Level Crossover Operator

With probability Pc, perform uniform crossover between test cases.

parent 1	x_1 = Test Case 1	x_2 = Test Case 2	x_3 = Test Case 4	x_5 = Test Case 5
parent 2	x_1 = Test Case 1	x_2 = Test Case 2	x_3 = Test Case 4	x_5 = Test Case 5
child 1	x_1 = Test Case 1	x_2 = Test Case 2	x_3 = Test Case 4	x_5 = Test Case 5

Heads = parent 1

Tails = parent 2

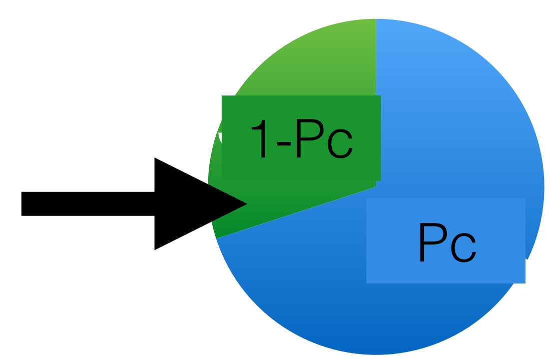


Higher Level Crossover Operator

With probability Pc, perform uniform crossover between Test Cases.

parent 1	x_1 = Test Case 1	x_2 = Test Case 2	x_3 = Test Case 4	x_5 = Test Case 5
parent 2	x_1 = Test Case 1	x_2 = Test Case 2	x_3 = Test Case 4	x_5 = Test Case 5

clone	x_1 = Test Case 1	x_2 = Test Case 2	x_3 = Test Case 4	x_5 = Test Case 5



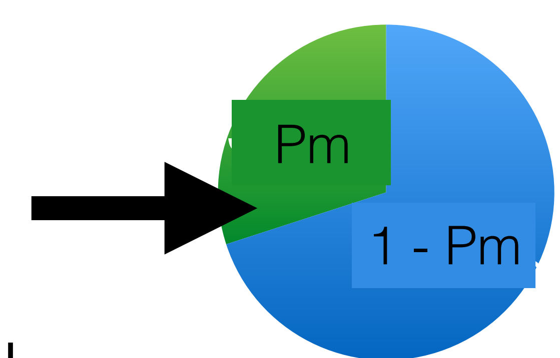
Higher Level Mutation Operator

With probability Pm, shuffle the order of the test cases.

individual

x_1 = Test Case 1	x_2 = Test Case 2	x_3 = Test Case 4	x_5 = Test Case 5

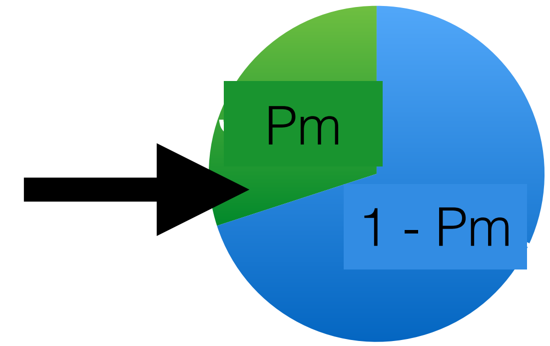
mutated individual

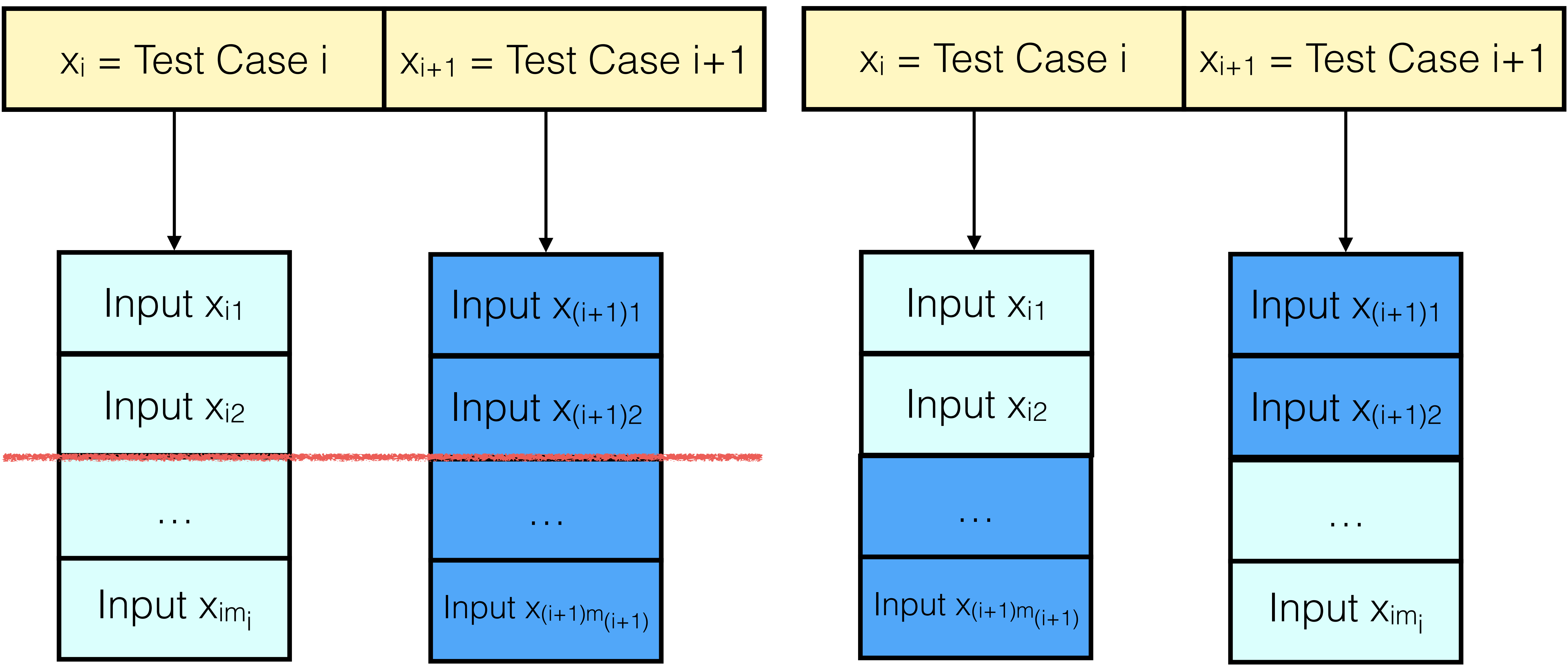


Only applied if Crossover is not applied.

Lower Level Mutation Operator 1

For each neighbouring test case **within an individual**, with probability Pm, perform 1-point crossover between them.

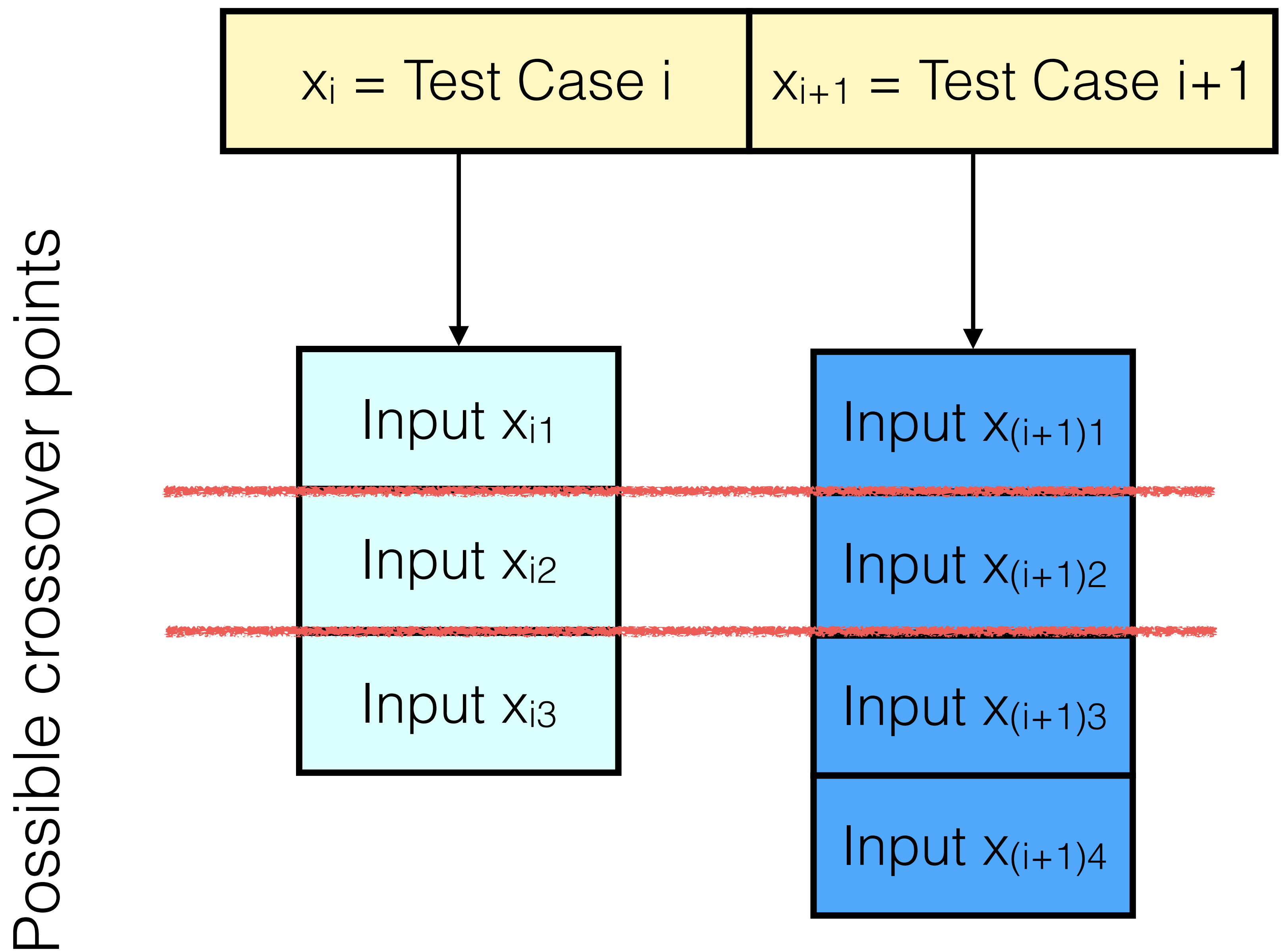




Only applied if Higher Level Mutation Operator is applied.

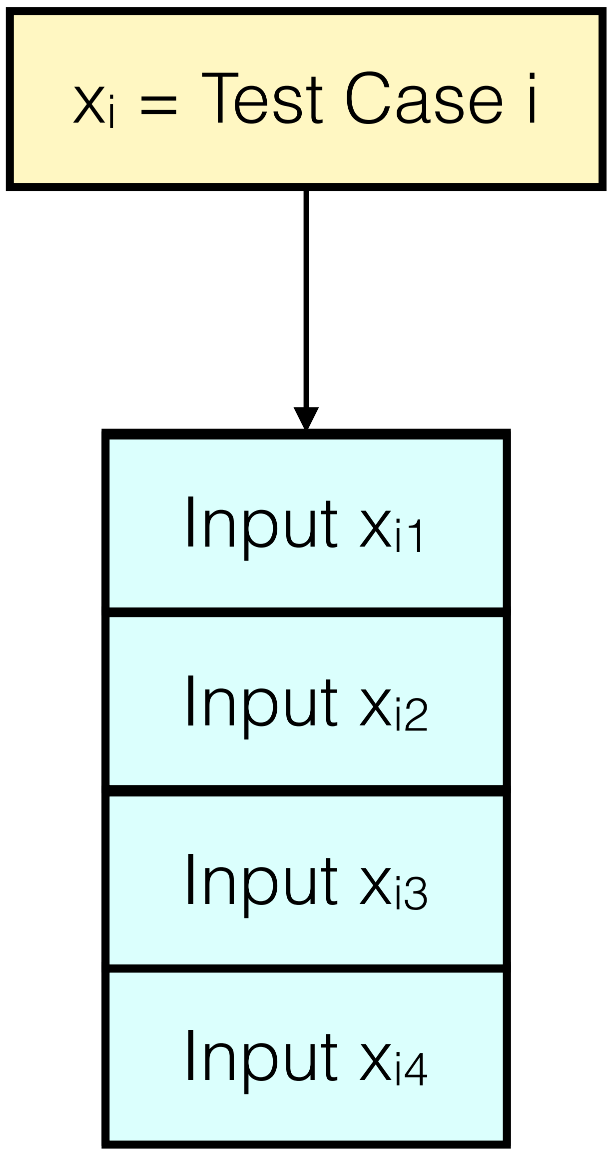
Lower Level Mutation Operator 1 — Test Cases of Different Sizes

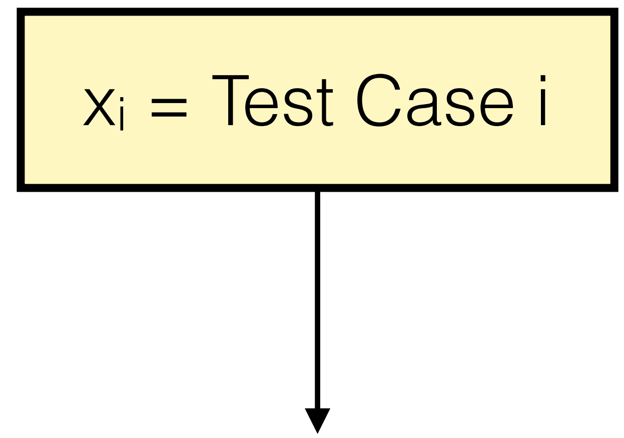
Crossover point is limited by the length of the shortest individual.

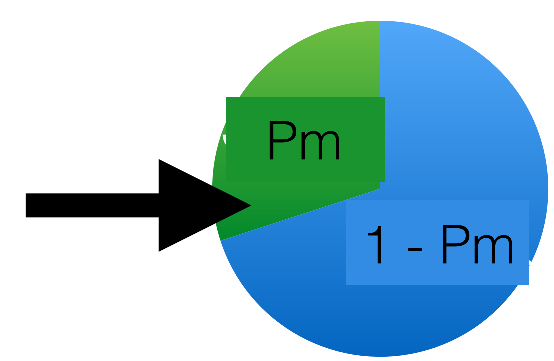


Lower Level Mutation Operator 2

For each test case within an individual,
with probability Pm, shuffle the order of the inputs within this test case.







Only applied if Higher Level Mutation Operator is applied.

Parents Selection

- Select 2 parents completely at random rather than tournament selection.
  - This is ok because survival selection still puts pressure towards better individuals in NSGA-II.

Problem Formulation

- Design variable: list with a given number of input sequences.
  - Design variable can be seen as a test suite.
  - Each input sequence is a test case with an expected “output” of “no crash”.
  - Examples of inputs for Android: Touch, Motion, Rotation, Track-ball, PinchZoom, Flip, Nav, MajorNav, AppSwitch, SysOp, enter text, or clicks on widgets.
- **Objectives:**
  - **Maximise coverage.**
  - Minimise length of test cases.
  - Maximise number of crashes found.
- Constraints:
  - N/A

Coverage Metrics

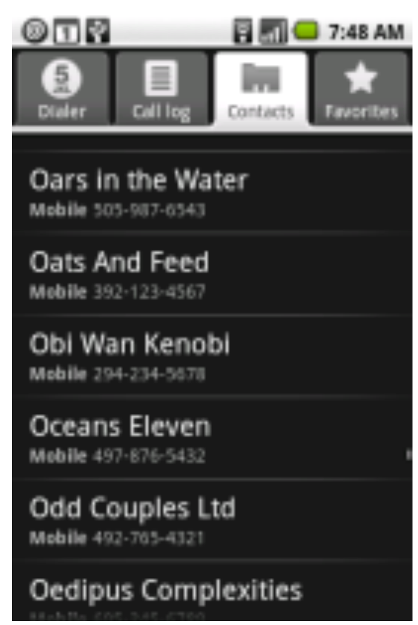
Different types of coverage are available:

- Statement coverage: number of statements exercised. (most fine grained)
- Method coverage: number of methods exercised.
- Android Activity coverage: number of Android Activities exercised. (coarser)

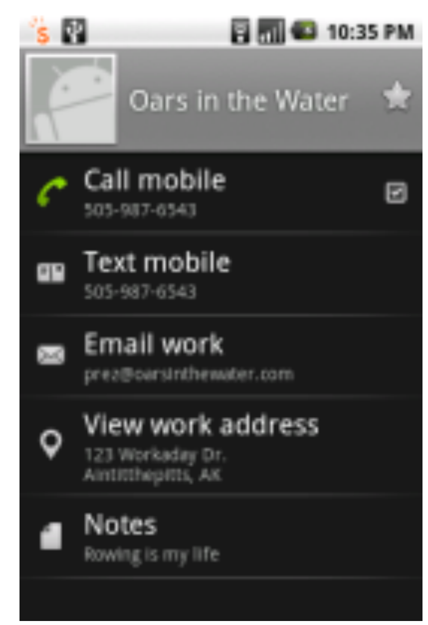
Activity Coverage

- Android Activities are the logical constructs of the screens that we want a user to navigate through.
- E.g., for a dialler app, you may have the following activities:

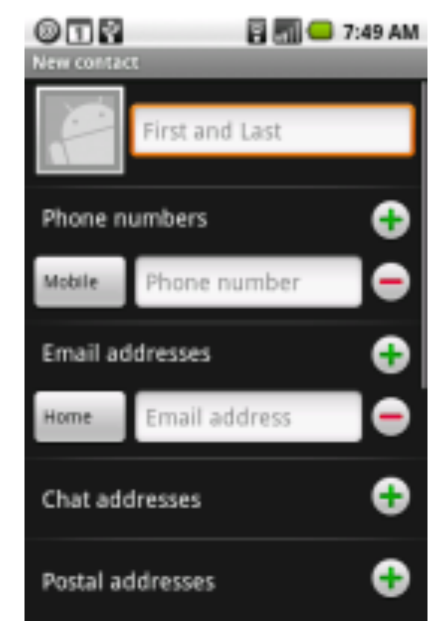
Dialer



Contacts



View Contact



New Contact

Coverage Metrics

In general, you can opt for more fine grained coverages if you have access to the code under test.

Otherwise, you must use a coarser coverage metric, such as Android Activity coverage.

Problem Formulation

- Design variable: list with a given number of input sequences.
  - Design variable can be seen as a test suite.
  - Each input sequence is a test case with an expected “output” of “no crash”.
  - Examples of inputs for Android: Touch, Motion, Rotation, Track-ball, PinchZoom, Flip, Nav, MajorNav, AppSwitch, SysOp, enter text, or clicks on widgets.

- **Objectives:**
  - Maximise coverage.
  - **Minimise length of test cases.**
  - Maximise number of crashes found.

- Constraints:
  - N/A

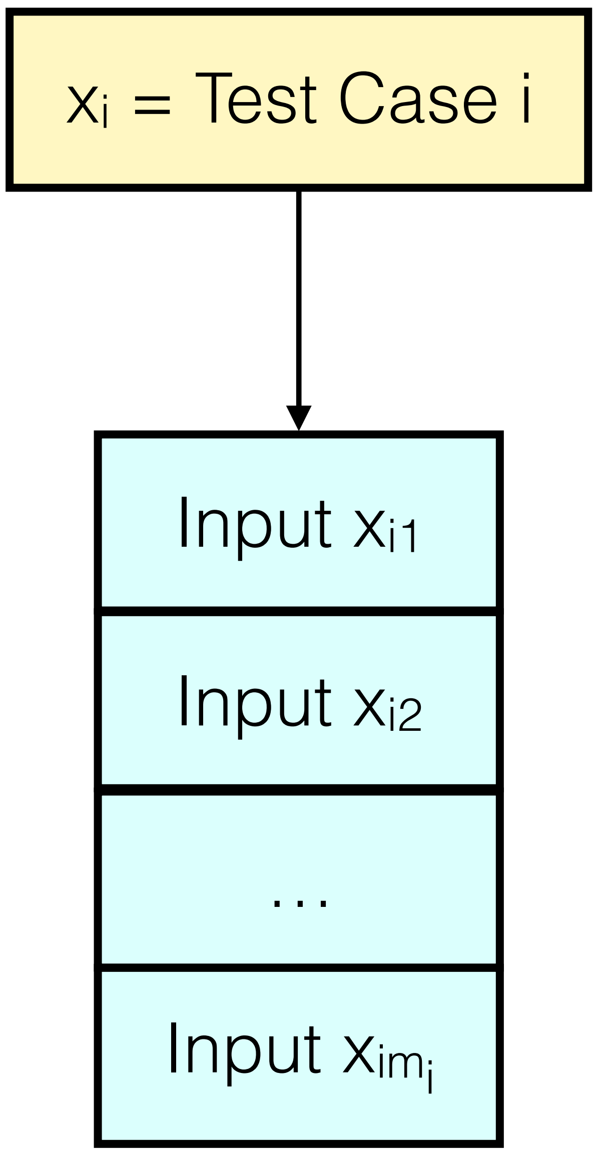
Length of Test Cases

Individual **x** (test suite):

x_1 = Test Case 1	x_2 = Test Case 2	…	x_n = Test Case *n*

$$\text{LengthIndividual}(\mathbf{x}) = \sum_{i=1}^{n} \text{LengthTestCase}(x_i)$$

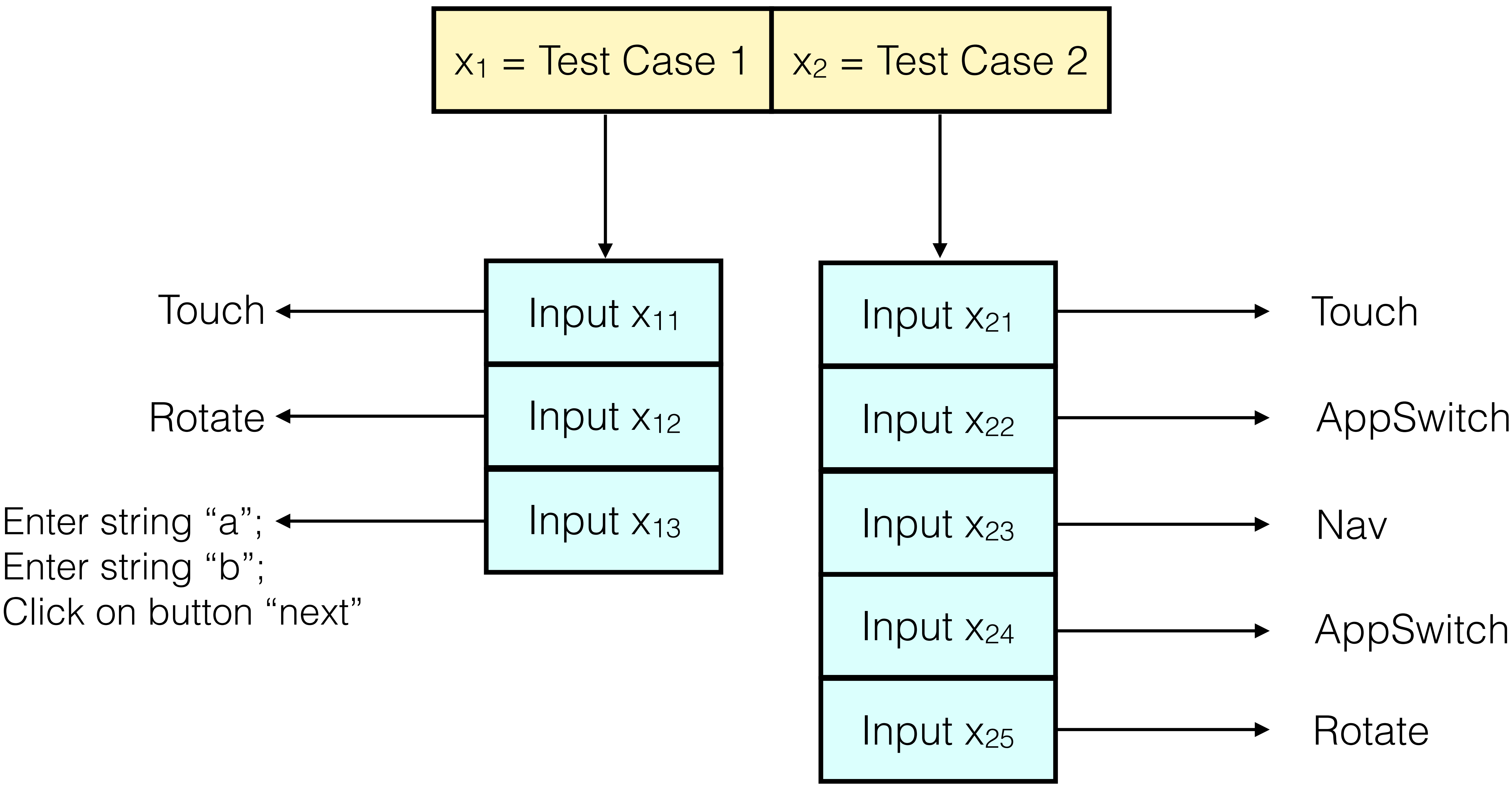
Length of a Single Test Case



$$\text{LengthTestCase}(x_i) = \sum_{j=1}^{m_i} \text{LengthInput}(x_{ij})$$

$$\text{LengthInput}(x_{ij}) = \begin{cases} 1, \text{ if } x_{ij} \text{ is an atomic event} \\ \text{number of strings + number of clicks, if } x_{ij} \text{ is a motif event} \end{cases}$$

Example of Calculation of Length



Problem Formulation

- Design variable: list with a given number of input sequences.
  - Design variable can be seen as a test suite.
  - Each input sequence is a test case with an expected “output” of “no crash”.
  - Examples of inputs for Android: Touch, Motion, Rotation, Track-ball, PinchZoom, Flip, Nav, MajorNav, AppSwitch, SysOp, enter text, or clicks on widgets.

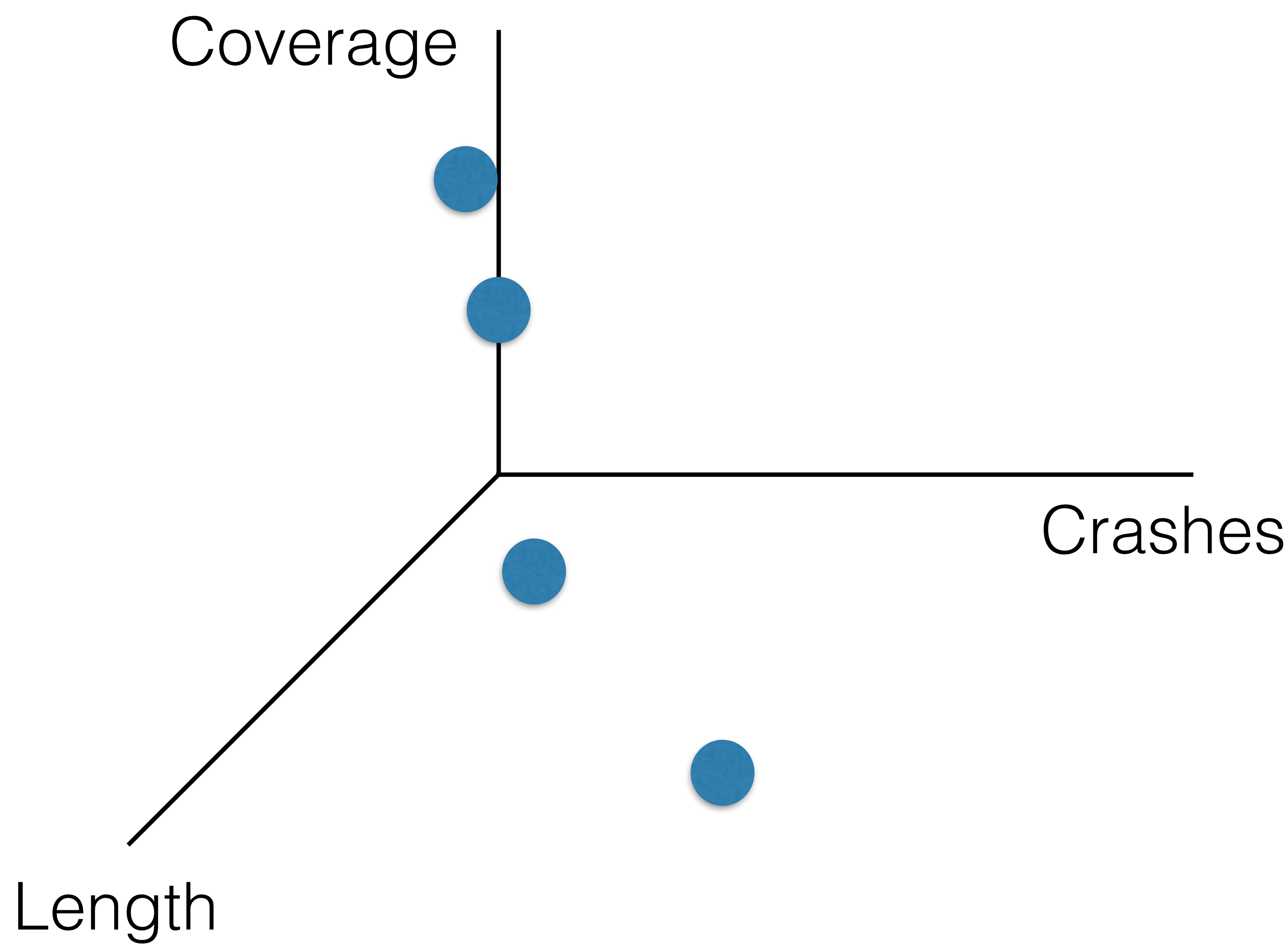
- **Objectives:**
  - Maximise coverage.
  - Minimise length of test cases.
  - **Maximise number of crashes found.**

- Constraints:
  - N/A

Number of Crashes Found

- Number of test cases that lead to a crash.

Non-Dominated Solutions



Software tester could choose 1+ individuals (test suites) to adopt as the final test suite.

An archive with all crash-inducing individuals can be kept.

Summary

- Software testing is time consuming.
- Evolutionary software testing can help to generate test suites.
- Formulation of test suite generation for finding crashes as an optimisation problem.
- NSGA-II design to solve this problem.

Further Reading

Sebastian Anthony

Facebook's evolutionary search for crashing software bugs

https://arstechnica.co.uk/information-technology/2017/08/facebook-dynamic-analysis-software-sapienz/

Ars Technika UK, 2017

Ke Mao, Mark Harman, Due Jia

Sapienz: Multi-objective Automated Testing for Android Applications
Proceedings of the 25th International Symposium on Software Testing and Analysis
Pages 94-105, 2016

https://dl.acm.org/citation.cfm?id=2931054