

Regression Testing

CS453, Shin Yoo

Overview

- What is “regression testing”? Why do we need one?
- Test Suite Minimisation
- Regression Test-case Selection
- Test Case Prioritisation
- Regression Testing and Continuous Integration

Regression Fault

Regression Fault

“After about 4 or 5 mins in a phone call my iPhone 3GS reboots itself for no apparent reason. This did not happen before I put the iOS4 update on it. Is there some way I can stop this happening, or revert back to the previous version?”

<http://discussions.apple.com/thread.jspa?threadID=2471090&tstart=0>

Regression Fault

“After about 4 or 5 mins in a phone call my iPhone 3GS reboots itself for no apparent reason. This did not happen before I put the iOS4 update on it. Is there some way I can stop this happening, or revert back to the previous version?”

<http://discussions.apple.com/thread.jspx?threadID=2471090&tstart=0>

“After reading a text message, i cant arrow back to next message. i have to close message program and reopen? iphone 6 problem began after ios11 install??”

<https://discussions.apple.com/thread/8083458>

Regression Fault

"After about 4 or 5 mins in a phone call my iPhone 3GS reboots itself for no apparent reason. This did not happen before I put the iOS4 update on it. Is there some way I can stop this happening, or revert back to the previous version?"

<http://discussions.apple.com/thread.jspx?threadID=2471090&tstart=0>

"After reading a text message, i cant arrow back to next message. i have to close message program and reopen? iphone 6 problem began after ios11 install?"

<https://discussions.apple.com/thread/8083458>

Regression Fault

- Regression fault occurs when new features or recent modifications to the software causes existing functionalities to break.

Regression Testing

- You are releasing a new version of your software.
- The new features have been thoroughly tested.
- You want to check if you have created any regression fault.
- How would you go about it?

Retest-all

Retest-all

- The simplest approach is called retest-all.

Retest-all

- The simplest approach is called retest-all.
- You just run all tests for the old features, not just ones for the new features.

Retest-all

- The simplest approach is called retest-all.
- You just run all tests for the old features, not just ones for the new features.
- There is one problem - can you guess what?

Retest-all

- The simplest approach is called retest-all.
- You just run all tests for the old features, not just ones for the new features.
- There is one problem - can you guess what?
- You eventually end up with too many tests and run out of time.

Inhibitive Cost

“For example, one of our industrial collaborators reports that for one of its products of about 20,000 lines of code, the entire test suite requires seven weeks to run.”

Prioritizing Test Cases for Regression Testing, IEEE Transactions on Software Engineering, G.Rothermel, R.H. Untch & M.J. Harrold (2001)

Inhibitive Cost



Long
Product
History

“For example, one of our industrial collaborators reports that for one of its products of about 20,000 lines of code, the entire test suite requires seven weeks to run.”

Prioritizing Test Cases for Regression Testing, IEEE Transactions on Software Engineering, G.Rothermel, R.H. Untch & M.J. Harrold (2001)

Inhibitive Cost



Long
Product
History



Different
Configurations

“For example, one of our industrial collaborators reports that for one of its products of about 20,000 lines of code, the entire test suite requires seven weeks to run.”

Prioritizing Test Cases for Regression Testing, IEEE Transactions on Software Engineering, G.Rothermel, R.H. Untch & M.J. Harrold (2001)

Inhibitive Cost



Long
Product
History



Different
Configurations



Types of
Test Cases

“For example, one of our industrial collaborators reports that for one of its products of about 20,000 lines of code, the entire test suite requires seven weeks to run.”

Prioritizing Test Cases for Regression Testing, IEEE Transactions on Software Engineering, G.Rothermel, R.H. Untch & M.J. Harrold (2001)

Regression Testing Techniques

- Many techniques have been developed in order to cope with the high cost of retest-all. They can be categorised into the following three categories:
- Test Suite Minimisation
- Regression Test-case Selection
- Test Case Prioritisation

Test Suite Minimisation

Test Suite Minimisation

- The Problem: Your regression test suite is too large.

Test Suite Minimisation

- The Problem: Your regression test suite is too large.
- The Idea: There must be some redundant test cases.

Test Suite Minimisation

- The Problem: Your regression test suite is too large.
- The Idea: There must be some redundant test cases.
- The Solution: Minimise (or reduce) your regression test suite by removing all the redundant tests.

Wait. “Redundant”?

Wait. “Redundant”?

- Redundant: “...could be omitted without loss of meaning or function...” (New Oxford American Dictionary)

Wait. “Redundant”?

- Redundant: “...could be omitted without loss of meaning or function...” (New Oxford American Dictionary)
- A test case can be labeled “redundant” only according to a specific criteria.

Wait. “Redundant”?

- Redundant: “...could be omitted without loss of meaning or function...” (New Oxford American Dictionary)
- A test case can be labeled “redundant” only according to a specific criteria.
- Remember DU-path? Statement coverage?

Test Suite Minimisation

Usually the information you need can be expressed as a matrix.

**Things to tick off
(branches, statements,
DU-paths, etc)**

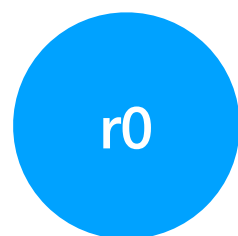
	r0	r1	r2	...
t0	1	1	0	
t1	0	1	0	
t2	0	0	1	
...				

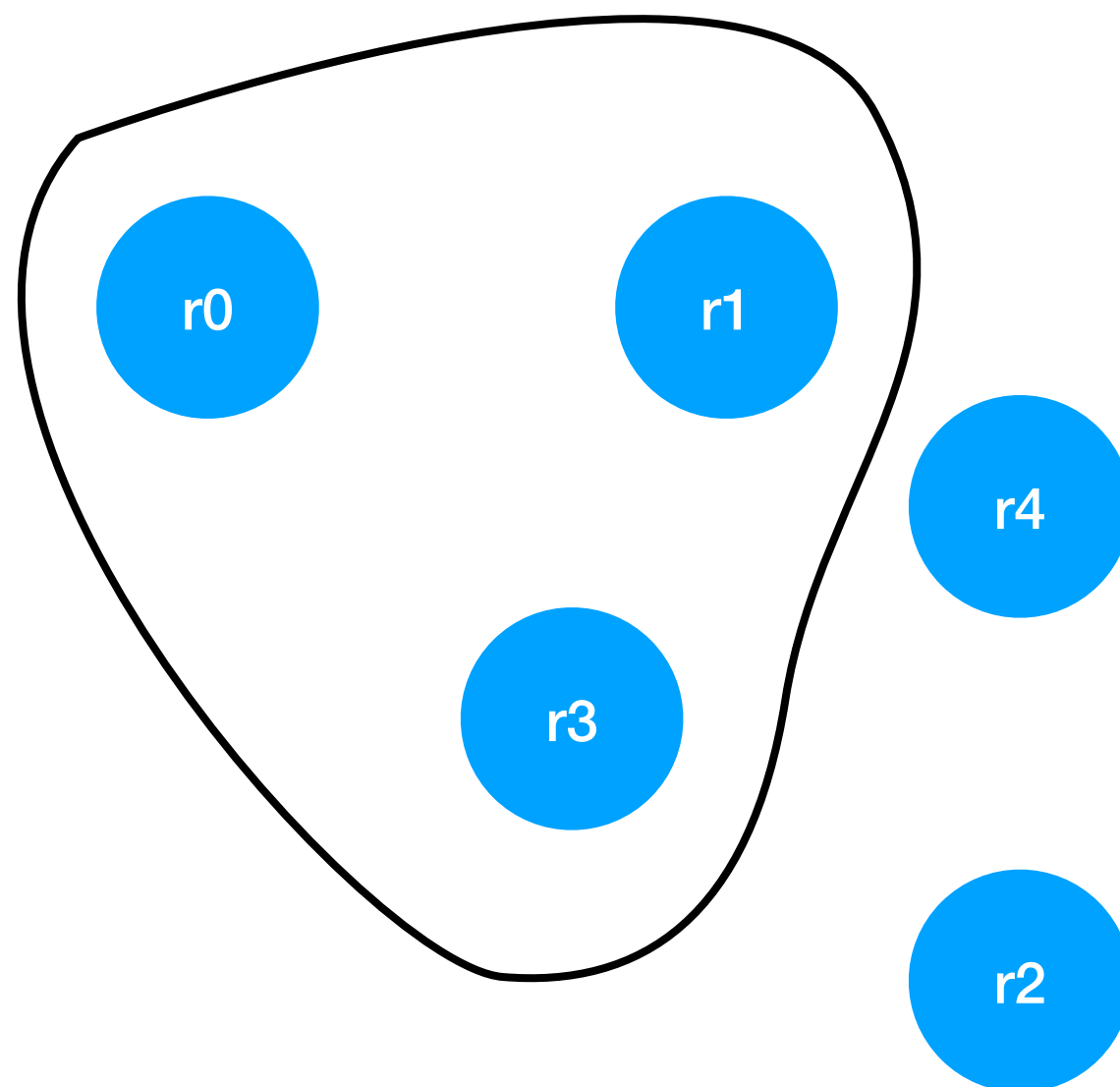


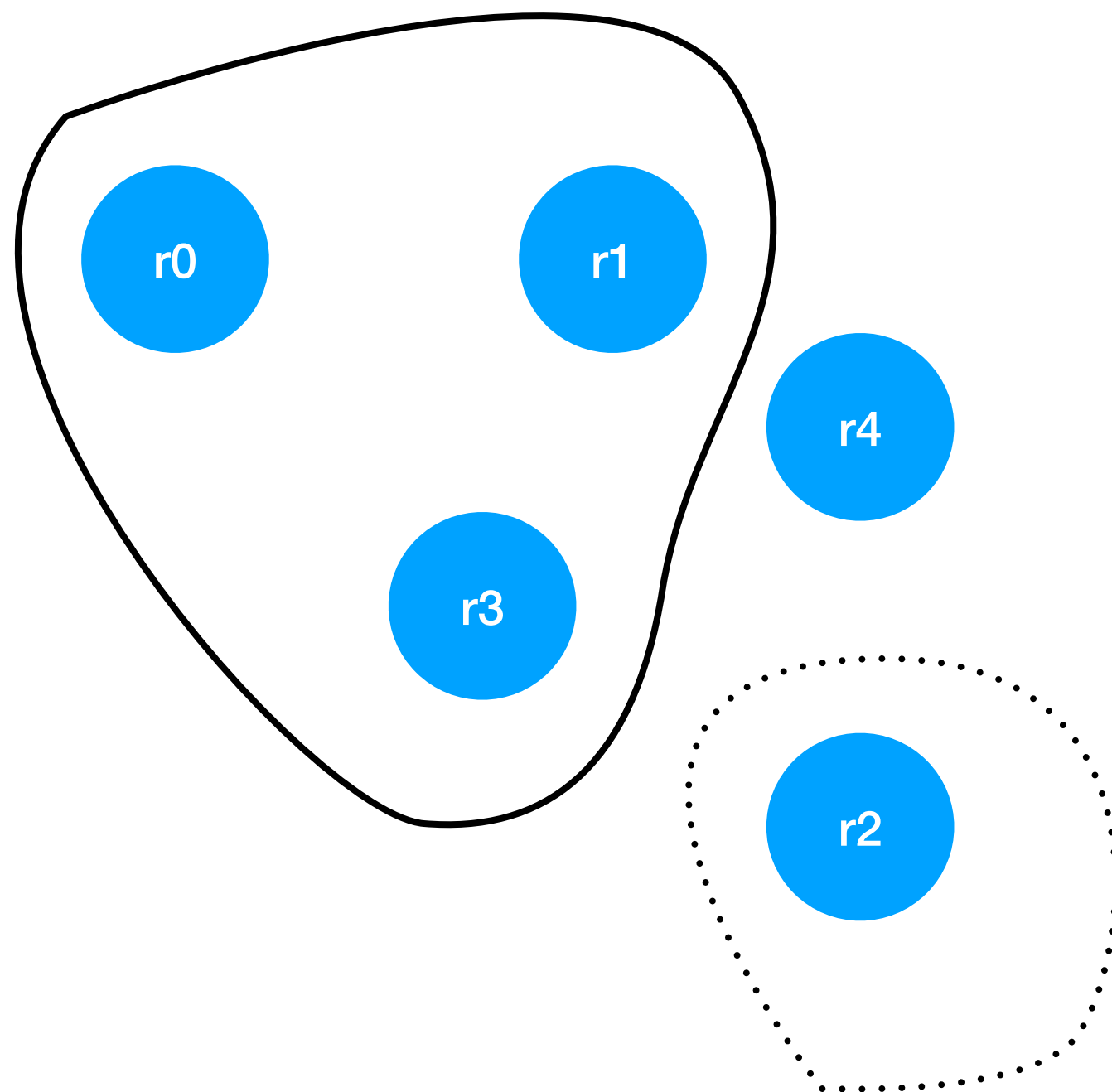
Now the problem becomes the following: what is the subset of rows (i.e. tests) that, when combined, will cover (i.e. put '1' on) the most of the columns (i.e. things to tick off)?

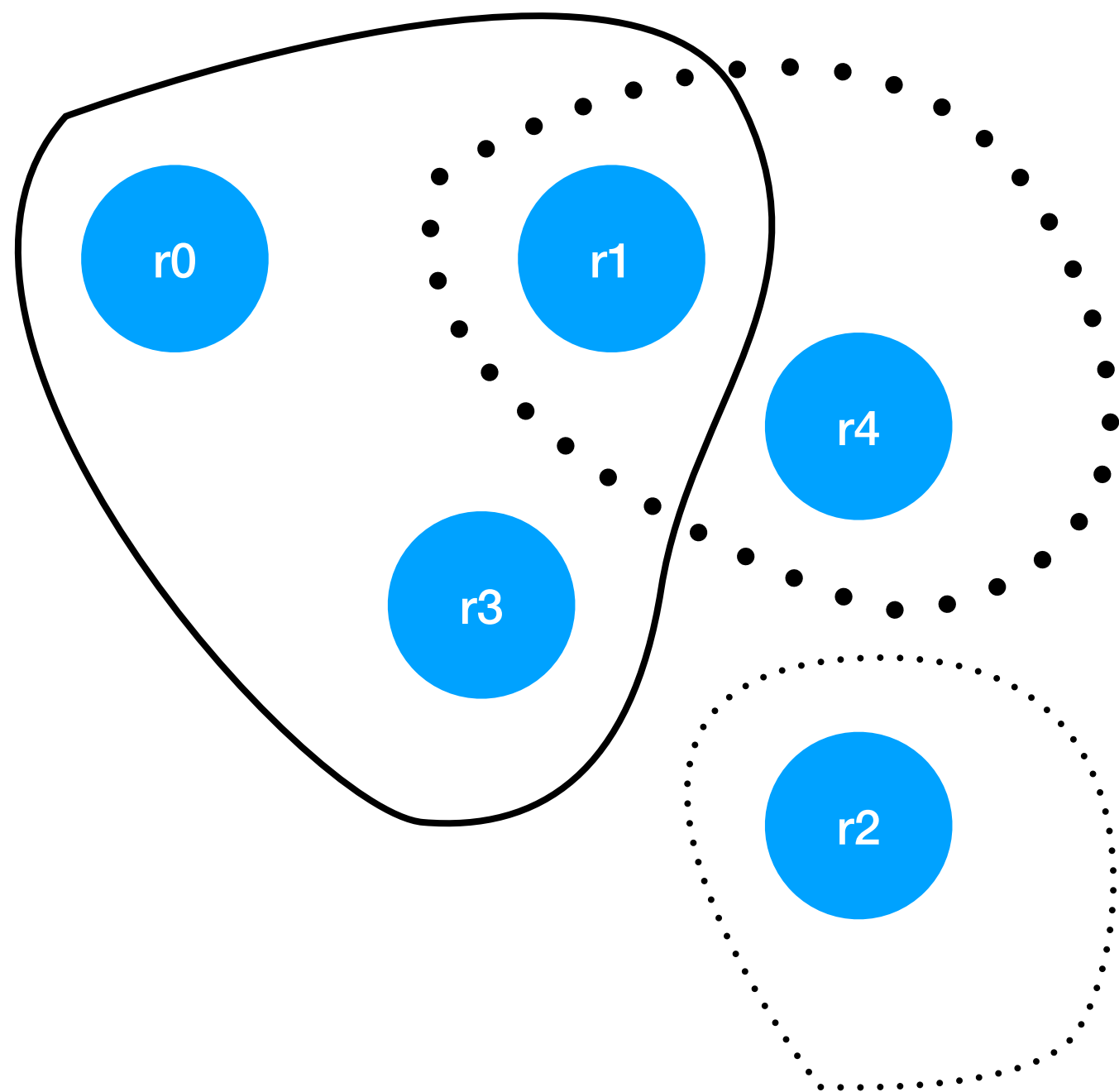
Test Suite Minimisation

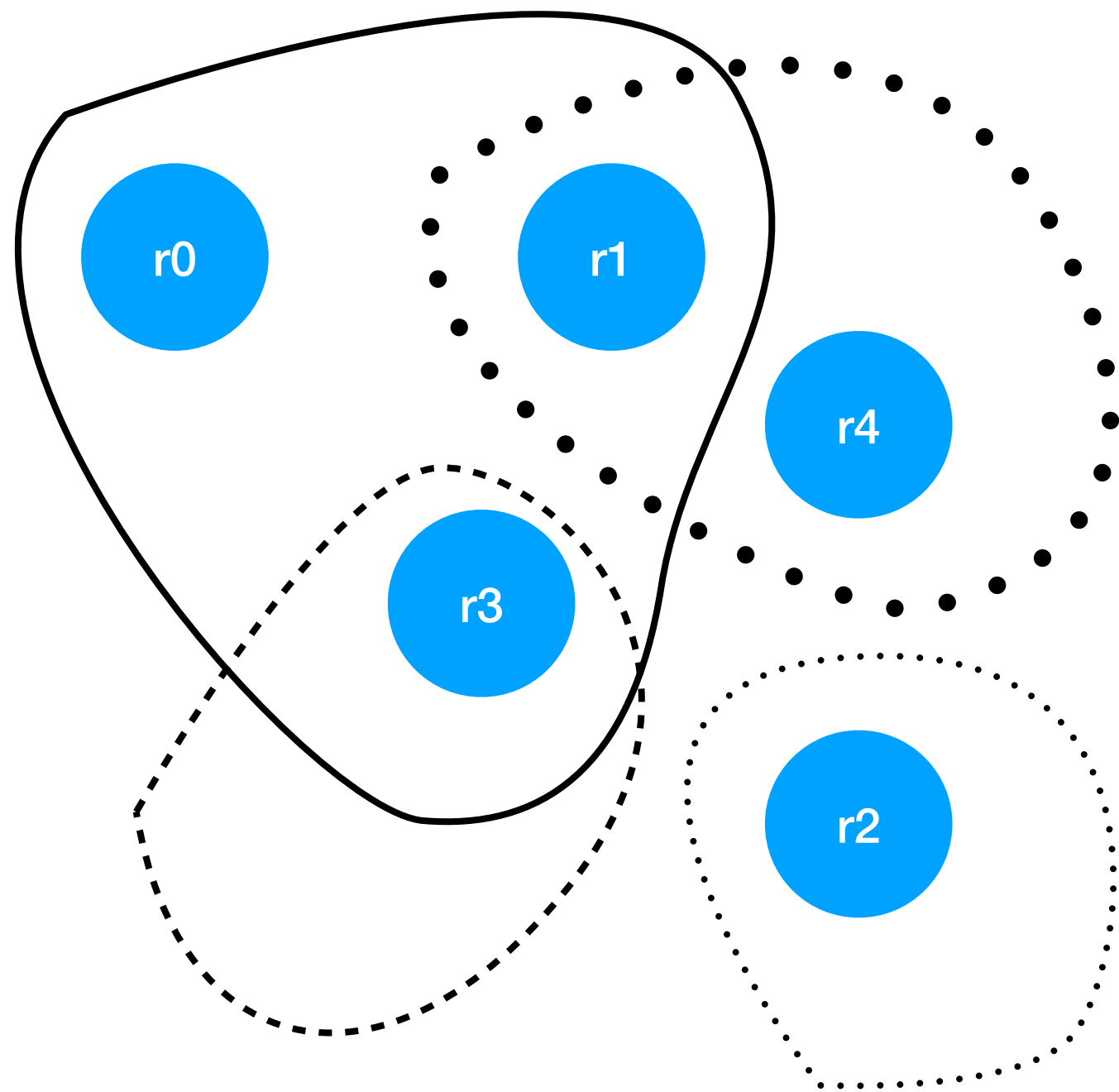
- The problem definition you just saw maps nicely into “set-cover” problem (http://en.wikipedia.org/wiki/Set_cover).
- Unfortunately, the problem is known to be NP-complete, meaning that there is no known efficient AND exact algorithm.
- We rely on approximated heuristics.

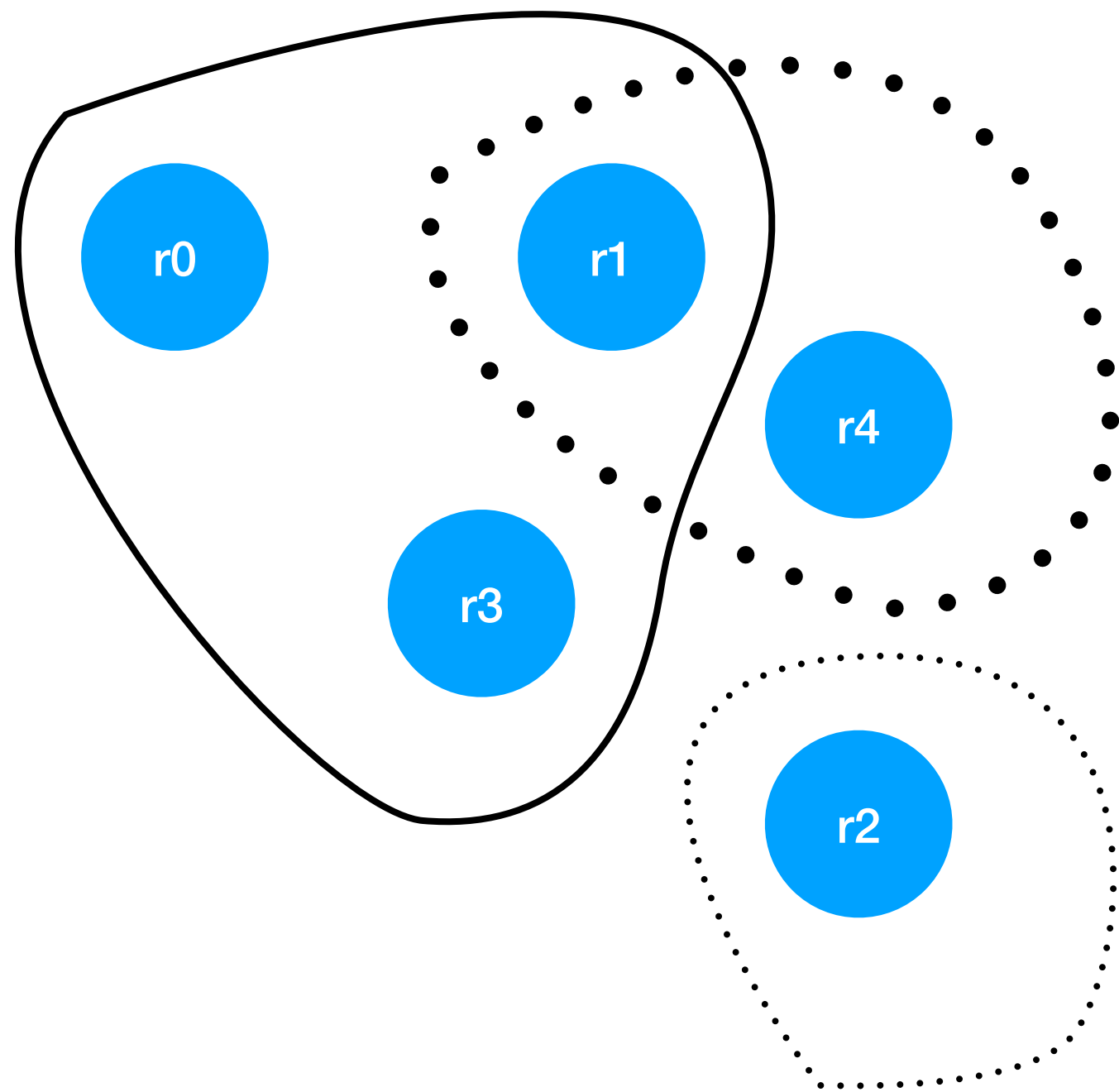












Test Suite Minimisation

- The problem definition you just saw maps nicely into “set-cover” problem (http://en.wikipedia.org/wiki/Set_cover).
- Unfortunately, the problem is known to be NP-complete, meaning that there is no known efficient AND exact algorithm.
- We rely on approximated heuristics.

Greedy Minimisation

```
greedy_minimisation(TestSuite T)
  S = {}
  while(True)
    find t in T s.t. S ∪ {t} covers max. cols.;
    if t exists:
      S = S ∪ {t}
      continue
    else:
      break
```

Greedy Minimisation

greedy_minimisation(TestSuite T)

S = {}

while(True)

find t in T s.t. S \cup {t} covers max. cols.;

if t exists:

S = S \cup {t}

continue

else:

break

	r0	r1	r2	r3
t0	1	1	0	0
t1	0	1	0	1
t2	0	0	1	1
t3	0	0	1	0

Greedy Minimisation

greedy_minimisation(TestSuite T)

S = {}

while(True)

find t in T s.t. S \cup {t} covers max. cols.;

if t exists:

S = S \cup {t}

continue

else:

break

	r0	r1	r2	r3
t0	1	1	0	0
t1	0	1	0	1
t2	0	0	1	1
t3	0	0	1	0

I0: S = {}

Greedy Minimisation

greedy_minimisation(TestSuite T)

S = {}

while(True)

find t in T s.t. S \cup {t} covers max. cols.;

if t exists:

S = S \cup {t}

continue

else:

break

	r0	r1	r2	r3
t0	1	1	0	0
t1	0	1	0	1
t2	0	0	1	1
t3	0	0	1	0

I0: S = {}

I1: S = {t0}

Greedy Minimisation

greedy_minimisation(TestSuite T)

S = {}

while(True)

find t in T s.t. S \cup {t} covers max. cols.;

if t exists:

S = S \cup {t}

continue

else:

break

	r0	r1	r2	r3
t0	1	1	0	0
t1	0	1	0	1
t2	0	0	1	1
t3	0	0	1	0

I0: S = {}

I1: S = {t0}

I2: S = {t0, t2}

Greedy Minimisation

greedy_minimisation(TestSuite T)

S = {}

while(True)

find t in T s.t. S \cup {t} covers max. cols.;

if t exists:

S = S \cup {t}

continue

else:

break

	r0	r1	r2	r3
t0	1	1	0	0
t1	0	1	0	1
t2	0	0	1	1
t3	0	0	1	0

I0: S = {}

I1: S = {t0}

I2: S = {t0, t2}

What about Cost?

	r0	r1	r2	r3	Time
t0	1	1	0	0	2
t1	0	1	0	1	3
t2	0	0	1	1	7
t3	0	0	1	0	3

- Suppose different test cases have different costs. For example, t2 takes 7 minutes to run, while t1 and t3 take 3 minutes each.
- Is it still sensible to minimise to {t0, t2}?

Greedy Minimisation w/ Cost

```
greedy_minimisation(TestSuite T)
  S = {}
  while(True)
    find t in T w/ max.  $\Delta\text{cov.}/\text{cost}$  of t;
    if t exists:
      S = S  $\cup$  {t}
      continue
    else:
      break
```

	r0	r1	r2	r3	Time
t0	1	1	0	0	2
t1	0	1	0	1	3
t2	0	0	1	1	7
t3	0	0	1	0	3

Greedy Minimisation w/ Cost

```
greedy_minimisation(TestSuite T)
  S = {}
  while(True)
    find t in T w/ max.  $\Delta\text{cov.}/\text{cost}$  of t;
    if t exists:
      S = S  $\cup$  {t}
      continue
    else:
      break
```

	r0	r1	r2	r3	Time
t0	1	1	0	0	2
t1	0	1	0	1	3
t2	0	0	1	1	7
t3	0	0	1	0	3

I0: S = {}

Greedy Minimisation w/ Cost

```
greedy_minimisation(TestSuite T)
  S = {}
  while(True)
    find t in T w/ max.  $\Delta\text{cov.}/\text{cost}$  of t;
    if t exists:
      S = S  $\cup$  {t}
      continue
    else:
      break
```

	r0	r1	r2	r3	Time
t0	1	1	0	0	2
t1	0	1	0	1	3
t2	0	0	1	1	7
t3	0	0	1	0	3

I0: S = {}
I1: S = {t0}

Greedy Minimisation w/ Cost

```
greedy_minimisation(TestSuite T)
  S = {}
  while(True)
    find t in T w/ max.  $\Delta\text{cov.}/\text{cost}$  of t;
    if t exists:
      S = S  $\cup$  {t}
      continue
    else:
      break
```

	r0	r1	r2	r3	Time
t0	1	1	0	0	2
t1	0	1	0	1	3
t2	0	0	1	1	7
t3	0	0	1	0	3

I0: S = {}
I1: S = {t0}
I2: S = {t0, t1}

Greedy Minimisation w/ Cost

```
greedy_minimisation(TestSuite T)
  S = {}
  while(True)
    find t in T w/ max.  $\Delta\text{cov.}/\text{cost}$  of t;
    if t exists:
      S = S  $\cup$  {t}
      continue
    else:
      break
```

	r0	r1	r2	r3	Time
t0	1	1	0	0	2
t1	0	1	0	1	3
t2	0	0	1	1	7
t3	0	0	1	0	3

```
I0: S = {}
I1: S = {t0}
I2: S = {t0, t1}
I3: S = {t0, t1, t3}
```

Multi-Objectiveness : Problems

- “After performing minimisation, the test suite is still too big. What can I actually do in the next 6 hours?”
- “I care not just code coverage, but something else too. Can I also achieve X with the minimised suite?”

Test Case	Program Blocks										Time
	1	2	3	4	5	6	7	8	9	10	
T1	x	x	x	x	x	x	x	x			4
T2	x	x		x	x	x	x	x	x	x	5
T3	x	x	x						x		3
T4	x	x	x	x	x						3



Test Case	Program Blocks										Time
	1	2	3	4	5	6	7	8	9	10	
T1	x	x	x	x	x	x	x	x			4
T2	x	x		x	x	x	x	x	x	x	5
T3	x	x	x						x		3
T4	x	x	x	x	x						3

Single Objective



Test Case	Program Blocks										Time
	1	2	3	4	5	6	7	8	9	10	
T1	x	x	x	x	x	x	x	x			4
T2	x	x		x	x	x	x	x	x	x	5
T3	x	x	x						x		3
T4	x	x	x	x	x						3

Single Objective

Choose test case with highest block per time ratio as the next one



Test Case	Program Blocks										Time
	1	2	3	4	5	6	7	8	9	10	
T1	x	x	x	x	x	x	x	x			4
T2	x	x		x	x	x	x	x	x	x	5
T3	x	x	x						x		3
T4	x	x	x	x	x						3

Single Objective

Choose test case with highest block per time ratio as the next one

- 1) T1 (ratio = 2.0)
- 2) T2 (ratio = 2 / 5 = 0.4)

$\therefore \{T1, T2\}$ (takes 9 hours)



Test Case	Program Blocks										Time
	1	2	3	4	5	6	7	8	9	10	
T1	x	x	x	x	x	x	x	x			4
T2	x	x		x	x	x	x	x	x	x	5
T3	x	x	x						x		3
T4	x	x	x	x	x						3

Single Objective

Choose test case with highest block per time ratio as the next one

1) T1 (ratio = 2.0)

2) T2 (ratio = 2 / 5 = 0.4)

∴ {T1, T2} (takes 9 hours)

“But we only have 7 hours...?”



Test Case	Program Blocks										Time
	1	2	3	4	5	6	7	8	9	10	
T1	x	x	x	x	x	x	x	x			4
T2	x	x		x	x	x	x	x	x	x	5
T3	x	x	x						x		3
T4	x	x	x	x	x						3

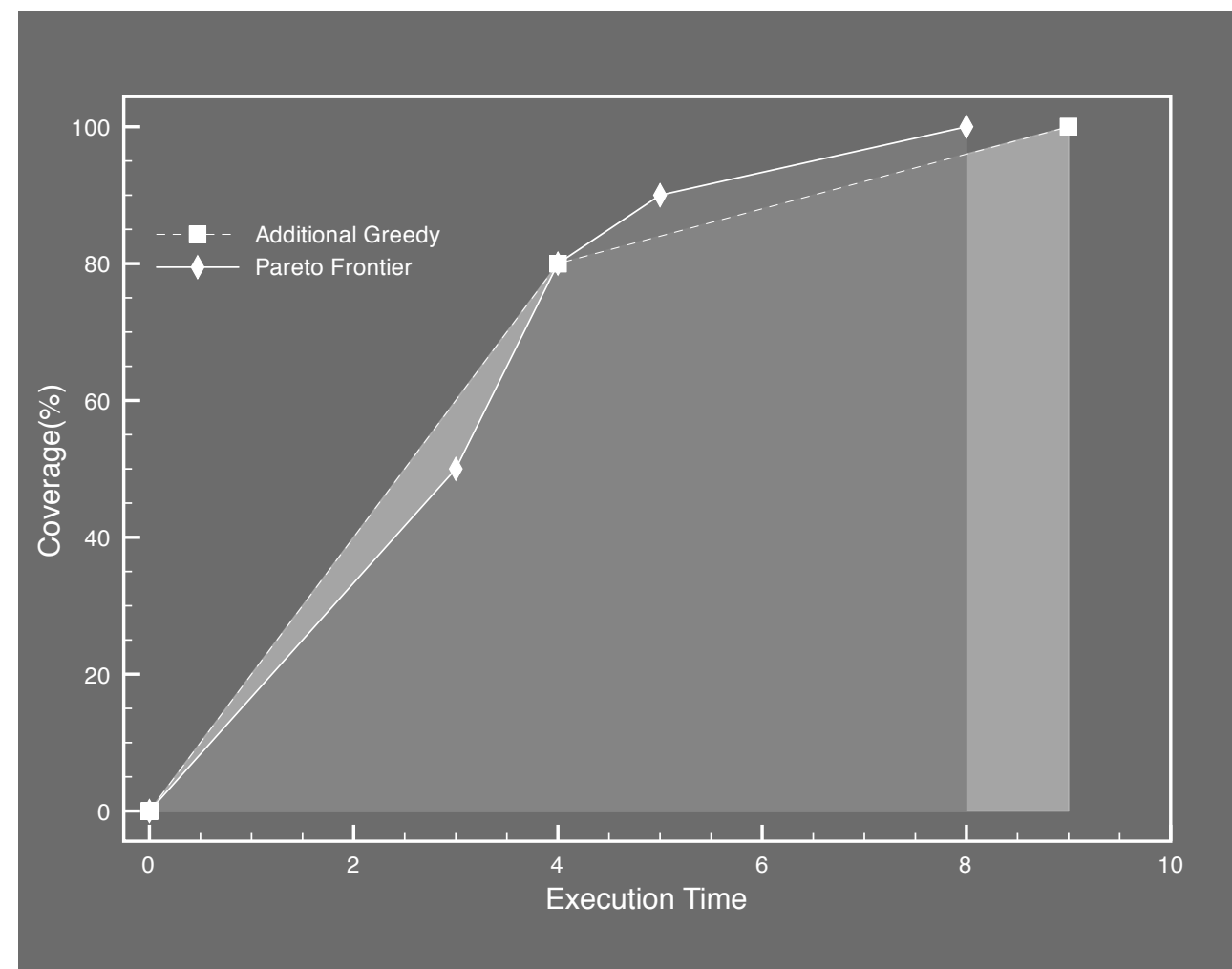
Single Objective

Choose test case with highest block per time ratio as the next one

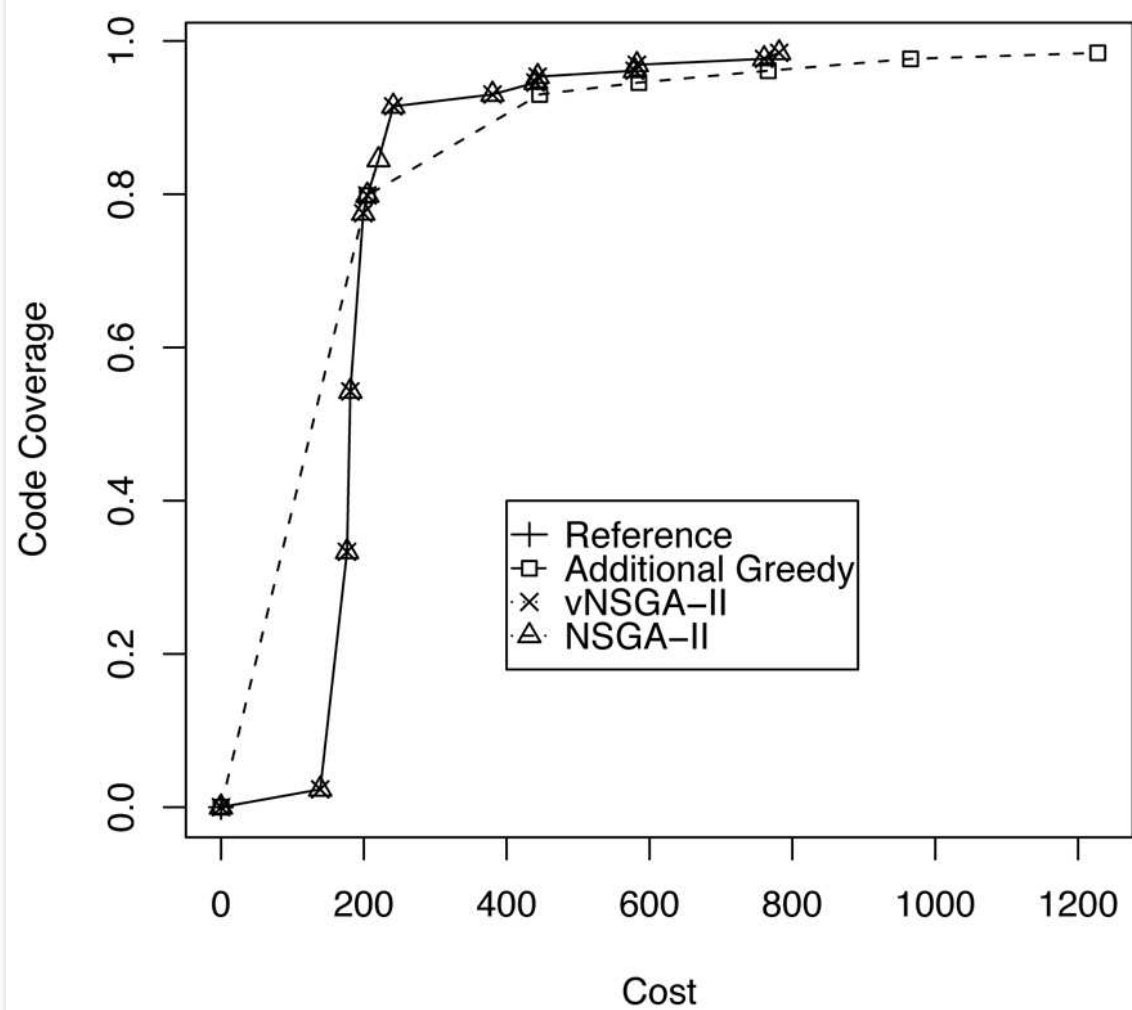
- 1) T1 (ratio = 2.0)
 - 2) T2 (ratio = 2 / 5 = 0.4)
- $\therefore \{T1, T2\}$ (takes 9 hours)

“But we only have 7 hours...?”

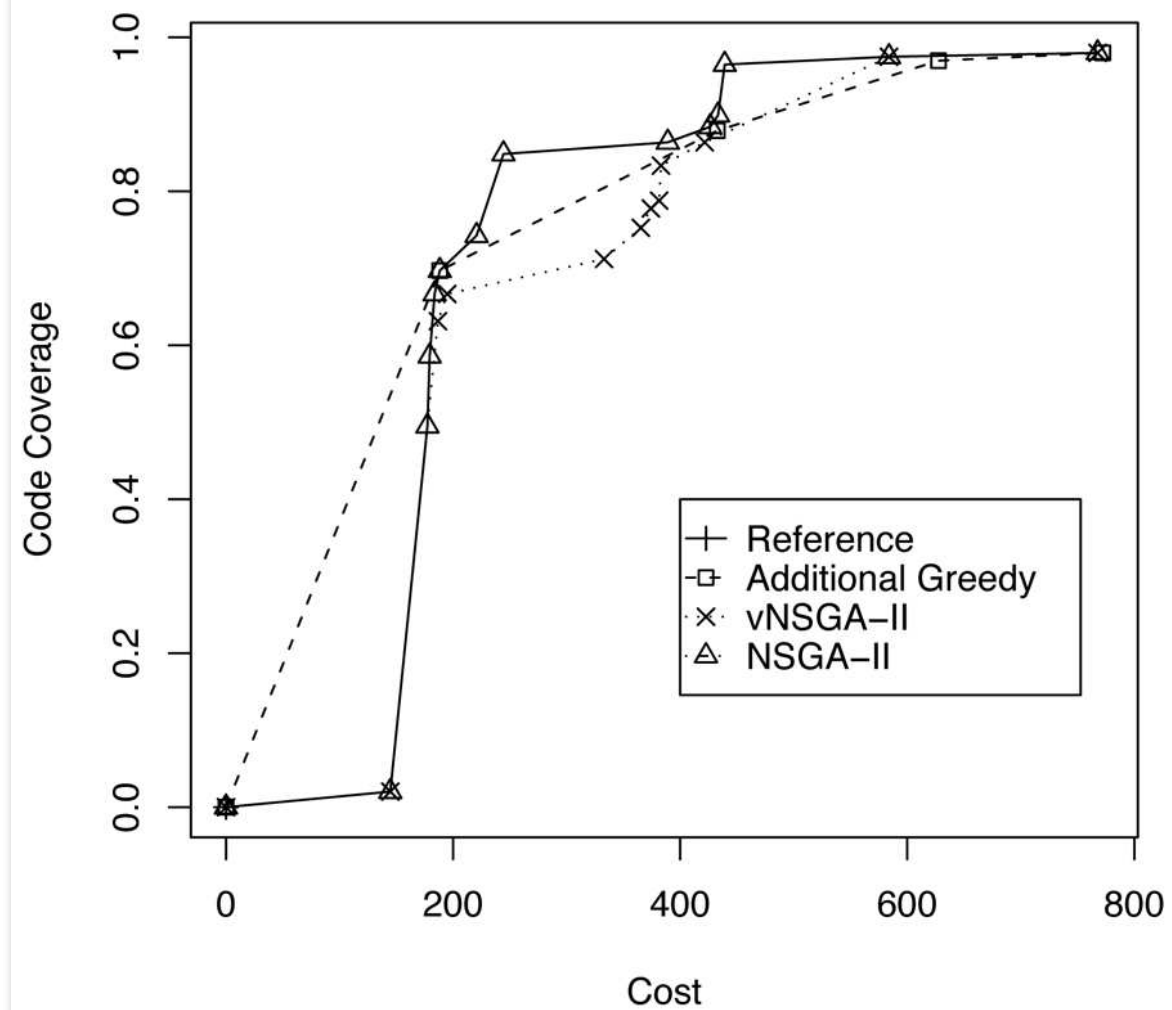
Multi Objective



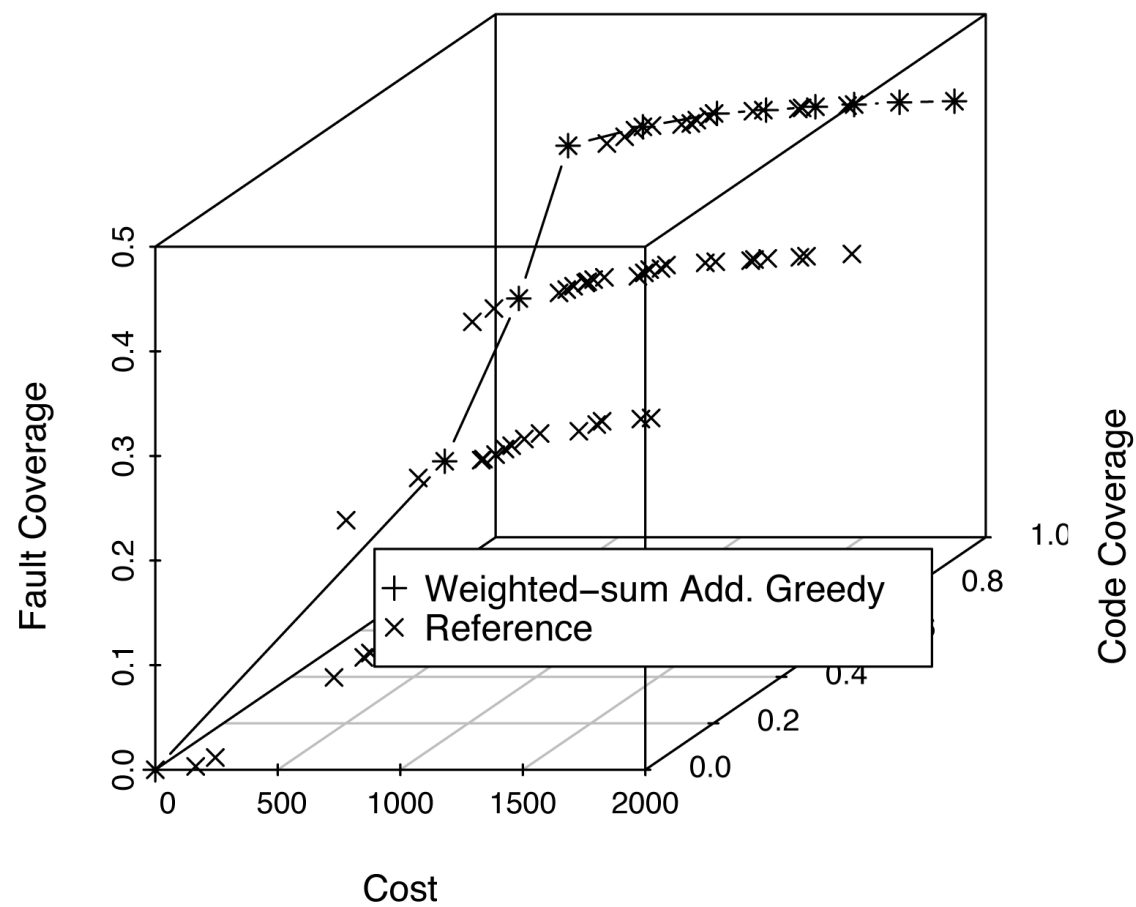
2 Objectives, schedule



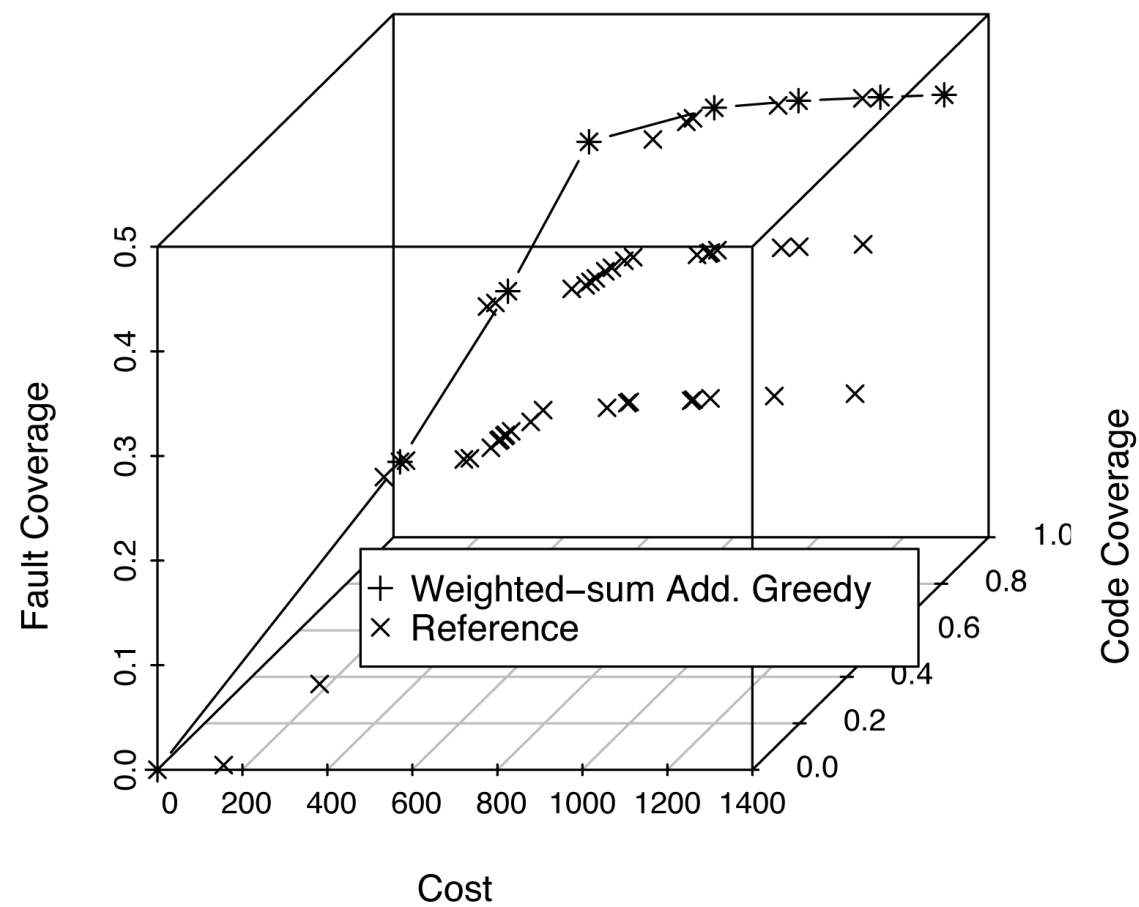
2 Objectives, printtokens2



printtokens



printtokens2



Regression Test-case Selection

Regression Test-case Selection

- The Problem: Your regression test suite is too large.

Regression Test-case Selection

- The Problem: Your regression test suite is too large.
- The Idea: Not all of your tests are related to the recent changes in software.

Regression Test-case Selection

- The Problem: Your regression test suite is too large.
- The Idea: Not all of your tests are related to the recent changes in software.
- The Solution: Precisely select only those tests that will actually execute the changed part of your software.

But how do we know?

- You have changed program P into P' .
- You have a test suite T ; you should also have “execution trace” of all the tests in T (i.e. a mapping from tests to statements/branches/etc that they execute) w.r.t. original P .
- Now pick your favourite methods, ‘cause there are tons :)

Textual Difference

```
20c20,21
< exit(0);
---
> printf("Illegal number of arguments\n");
> exit(1);
```

- Just use stock Unix tool, diff
- Line 20 was modified into 2 lines; execute all tests that includes line 20 in their trace.
- Simple!

Graph Walking

P

```
void test_me(int x)
{
    if(x == 0){
        print "Bwahaha";
    }
    return;
}
```

P'

```
void test_me_again(int x)
{
    if(x == 0){
        print "Bwahaha";
        print x;
    }
    return;
}
```

- Uses Control Flow Graph (CFG) that you already learned.

Test Suite

t0: x = 0

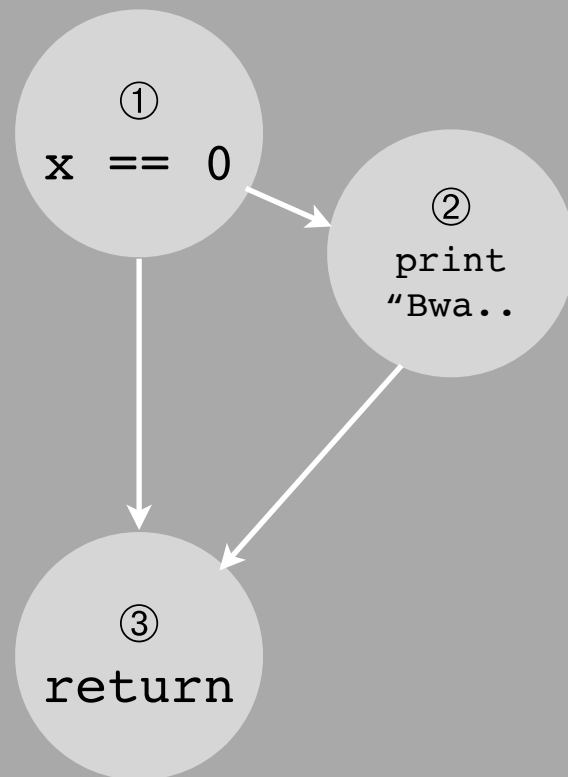
t1: x = 1

Which test would you select to test P→P'?

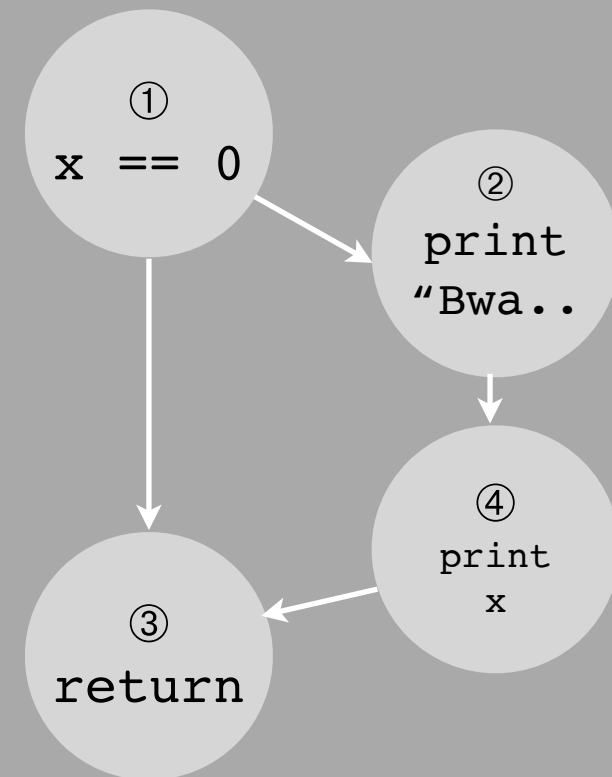
A safe and efficient algorithm for regression test selection, ICSM, Rothermel & Harrold, 1993

Graph Walking (Rothermel & Harrold, 1993)

P



P'



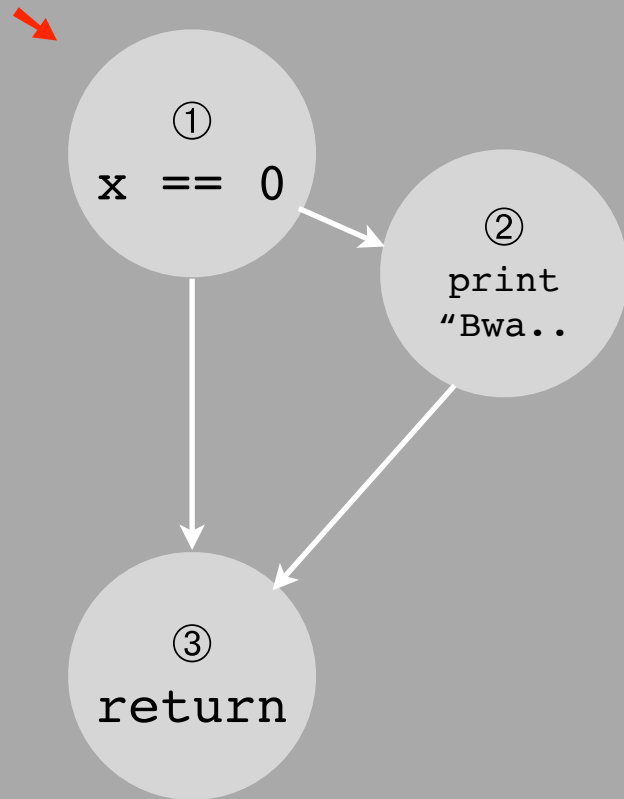
Test Suite

t0: x = 0 (①, ②, ③)

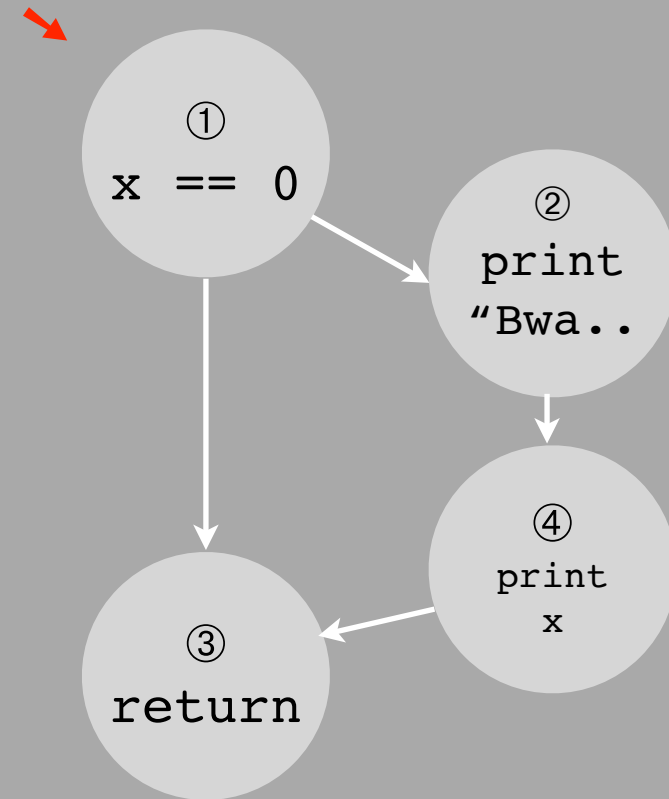
t1: x = 1 (①, ③)

Graph Walking (Rothermel & Harrold, 1993)

P



P'



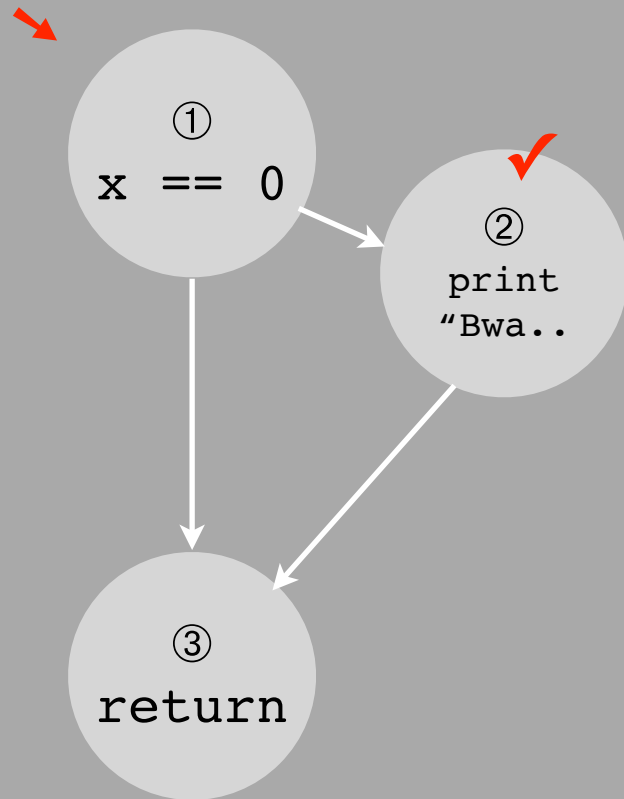
Test Suite

t0: `x = 0` (①, ②, ③)

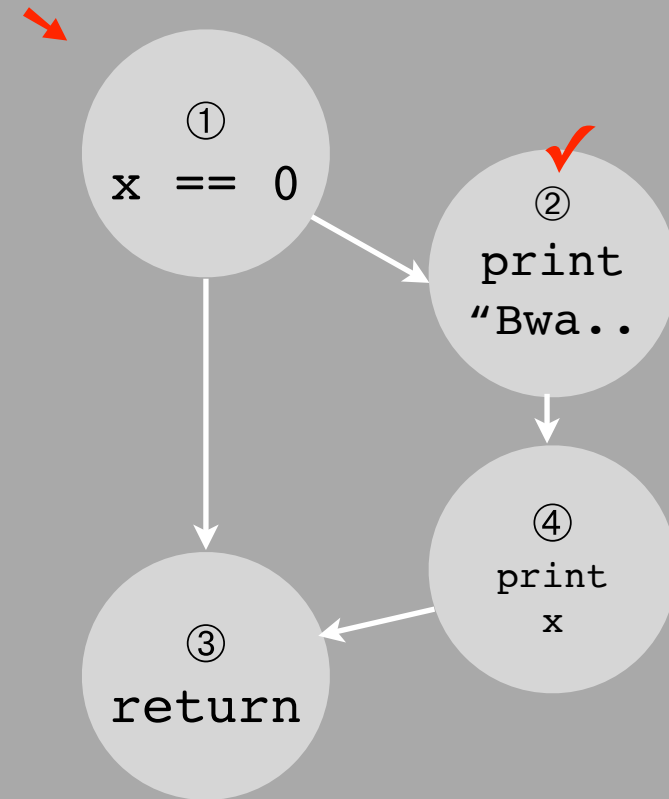
t1: `x = 1` (①, ③)

Graph Walking (Rothermel & Harrold, 1993)

P



P'



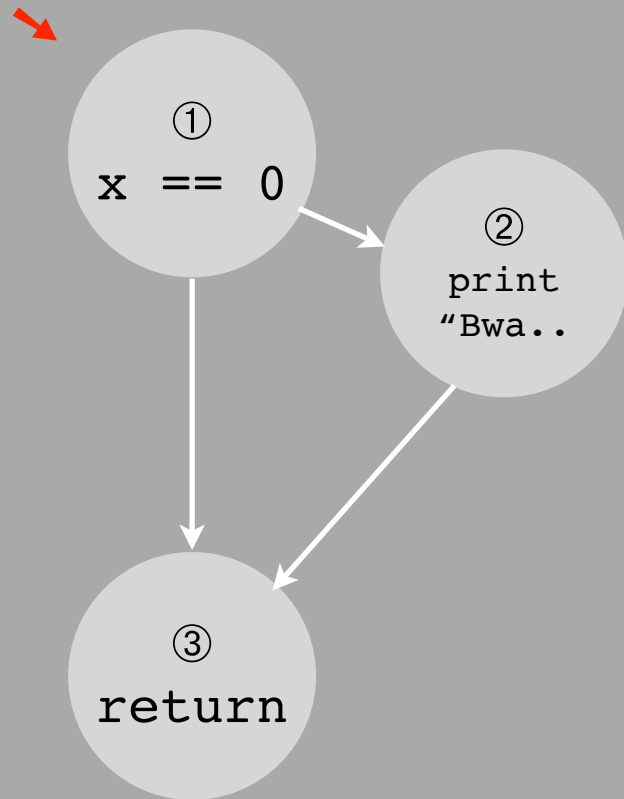
Test Suite

t0: `x = 0` (①, ②, ③)

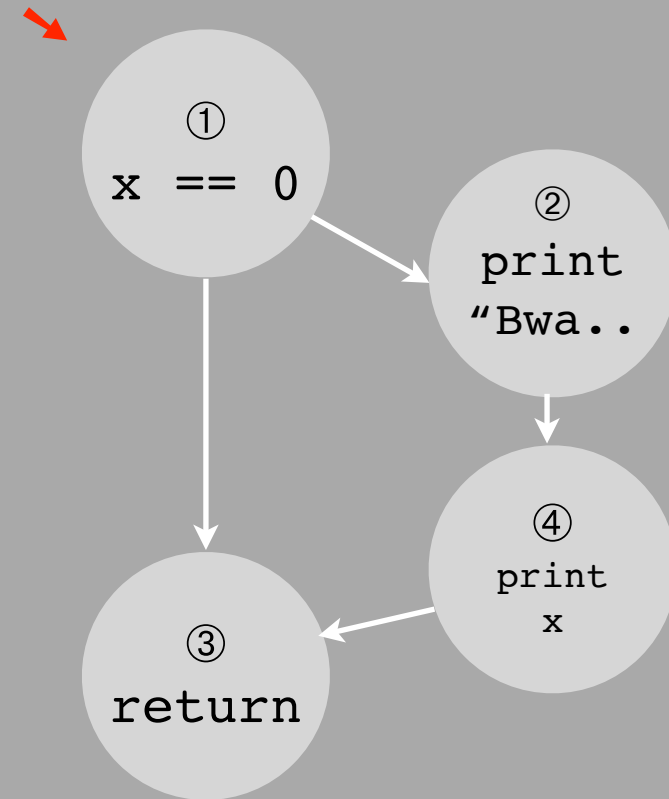
t1: `x = 1` (①, ③)

Graph Walking (Rothermel & Harrold, 1993)

P



P'



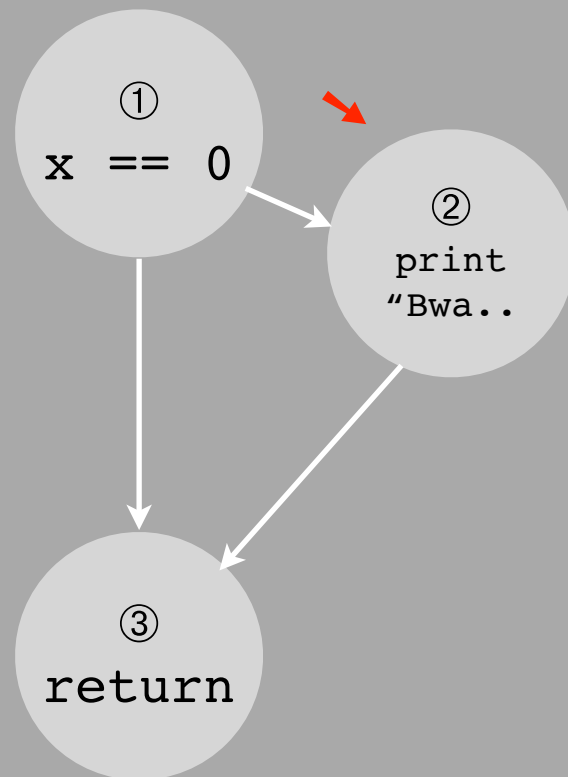
Test Suite

t0: `x = 0` (①, ②, ③)

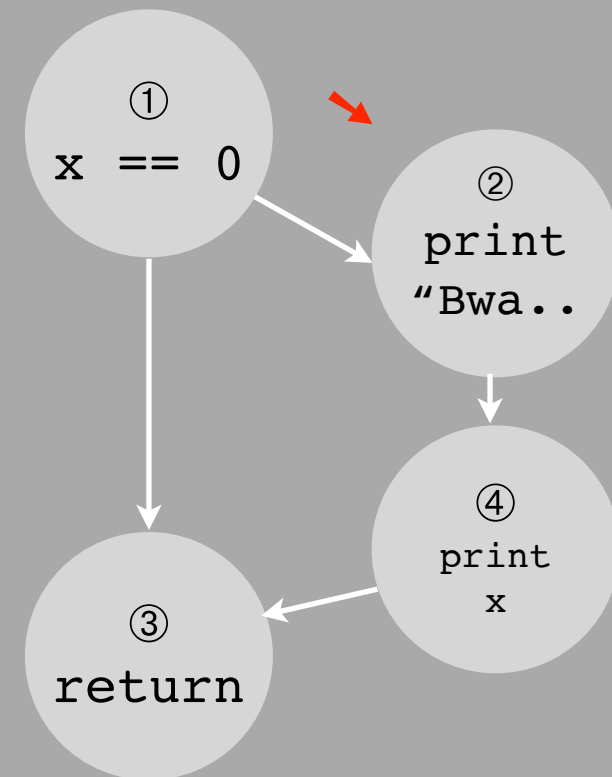
t1: `x = 1` (①, ③)

Graph Walking (Rothermel & Harrold, 1993)

P



P'



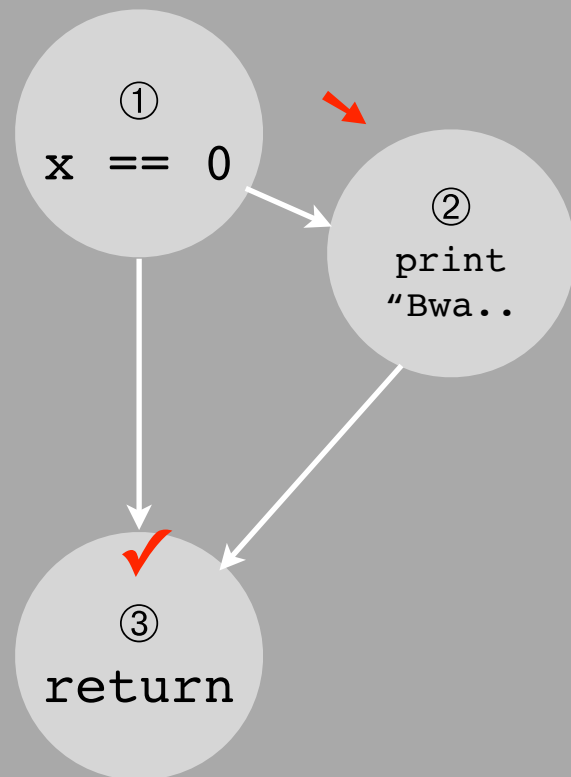
Test Suite

t0: `x = 0` (①, ②, ③)

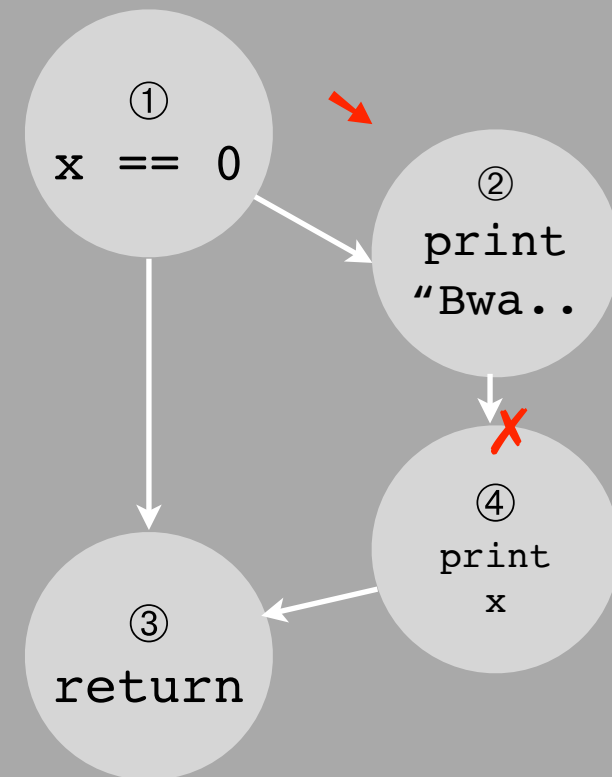
t1: `x = 1` (①, ③)

Graph Walking (Rothermel & Harrold, 1993)

P



P'



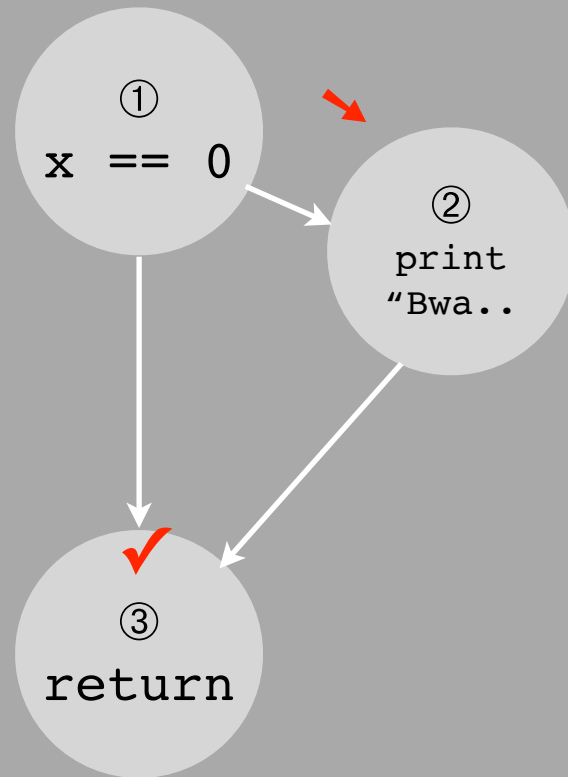
Test Suite

t0: `x = 0` (①, ②, ③)

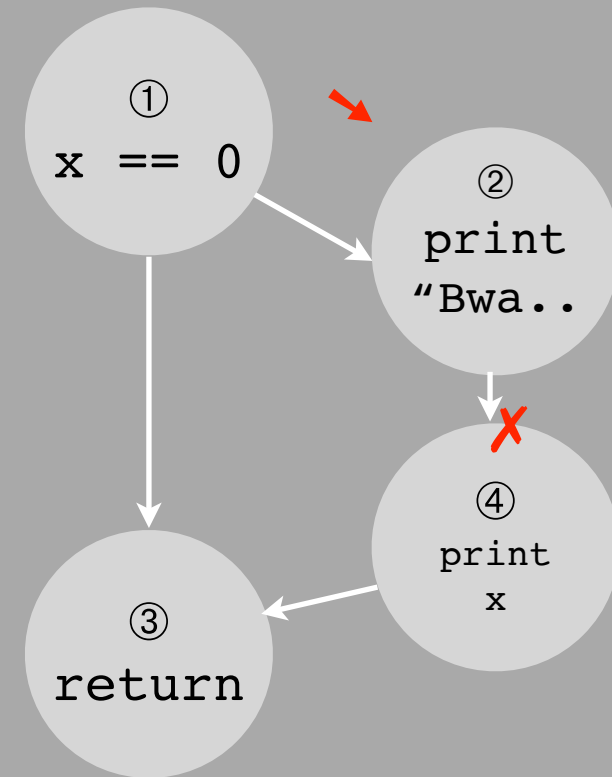
t1: `x = 1` (①, ③)

Graph Walking (Rothermel & Harrold, 1993)

P



P'



Test Suite

t0: `x = 0` (①, ②, ③)

t1: `x = 1` (①, ③)

P and P' start to differ from ②.

Therefore, any test that execute ② should be selected to test the change.

We select {t0}.

And many more

- Design artefacts (UML), data-flow analysis, path analysis, symbolic execution and many other techniques have been applied to aid precise selection of tests.

Safe Selection

- Selection techniques do not consider the cost of tests.
Why?
- Because they try to be “safe”, i.e., try to execute ALL tests that have the remotest possibility of revealing a fault related to the recent modification.
- Realistically, by “the remotest possibility” we mean that the test executes the modified parts of the program; we do not know any more than that!

Difficulties

- Data collection: collecting execution traces can be a hard task - especially if your system is big and consists of several different languages.
- Non-executable modifications: things like configuration changes can be tricky to analyse.
- Safety can be expensive: what if your safe selection is still too expensive to run?

Test Case Prioritisation

Test Case Prioritisation

- The Problem: Your regression test suite is too large.

Test Case Prioritisation

- The Problem: Your regression test suite is too large.
- The Idea: Let's execute tests following the order of importance.

Test Case Prioritisation

- The Problem: Your regression test suite is too large.
- The Idea: Let's execute tests following the order of importance.
- The Solution: Prioritise your test suite so that you get the most out of your regression testing whenever it gets stopped.

A Simple Tip

- You built your test suite by adding new test case whenever you added new feature: they are ordered from t1 to t100.
- Without using any complicated technique at all, which order will you execute your test cases in?

A Simple Tip

- You built your test suite by adding new test case whenever you added new feature: they are ordered from t1 to t100.
- Without using any complicated technique at all, which order will you execute your test cases in?
 - **Backwards**: newer faults are more likely to be detected by newer test.

A Simple Tip

- You built your test suite by adding new test case whenever you added new feature: they are ordered from t1 to t100.
- Without using any complicated technique at all, which order will you execute your test cases in?
 - **Backwards**: newer faults are more likely to be detected by newer test.
 - Random: choose the next test **randomly**.

A Simple Tip

- You built your test suite by adding new test case whenever you added new feature: they are ordered from t1 to t100.
- Without using any complicated technique at all, which order will you execute your test cases in?
 - **Backwards**: newer faults are more likely to be detected by newer test.
 - Random: choose the next test **randomly**.
- Both have good chance of being better than the given order (0 to 100).

Test Case Prioritisation

	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9
t0	x				x					
t1	x				x	x	x			
t2	x	x	x	x	x	x	x			
t3					x					
t4								x	x	x

- Suppose we knew about all the faults in advance (impossible, yes, but let's pretend).

Test Case Prioritisation

	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9
t0	x				x					
t1	x				x	x	x			
t2	x	x	x	x	x	x	x			
t3					x					
t4								x	x	x

- Suppose we knew about all the faults in advance (impossible, yes, but let's pretend).
- Which test would you run first if you knew this? What next?

Test Case Prioritisation

	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9
t0	x				x					
t1	x				x	x	x			
t2	x	x	x	x	x	x	x			
t3					x					
t4								x	x	x

- Suppose we knew about all the faults in advance (impossible, yes, but let's pretend).
- Which test would you run first if you knew this? What next?
- Obviously, it is t2 followed by t4.

Surrogate

- Again, we do not know about all the faults in advance; therefore, the ideal ordering (one that finds all known faults as soon as possible) is not feasible.
- Instead, we maximise the early realisation of some surrogate.
- Most common one is coverage; the assumption is that, if we cover the program as much/as early as possible, we will also maximise the chance of early fault detection.

Surrogate

	s0	s1	s2	s3	s4	s5	s6	s7	s8	s9
t0	x				x					
t1	x				x	x	x			
t2	x	x	x	x	x	x	x			
t3					x					
t4								x	x	x

- It does not change things much; instead of trying to detect each fault, we try to cover each statement(branch/block/etc).

Surrogate

	s0	s1	s2	s3	s4	s5	s6	s7	s8	s9
t0	x				x					
t1	x				x	x	x			
t2	x	x	x	x	x	x	x			
t3					x					
t4								x	x	x

- It does not change things much; instead of trying to detect each fault, we try to cover each statement(branch/block/etc).
- The ideal ordering still begins with t2-t4.

Surrogate

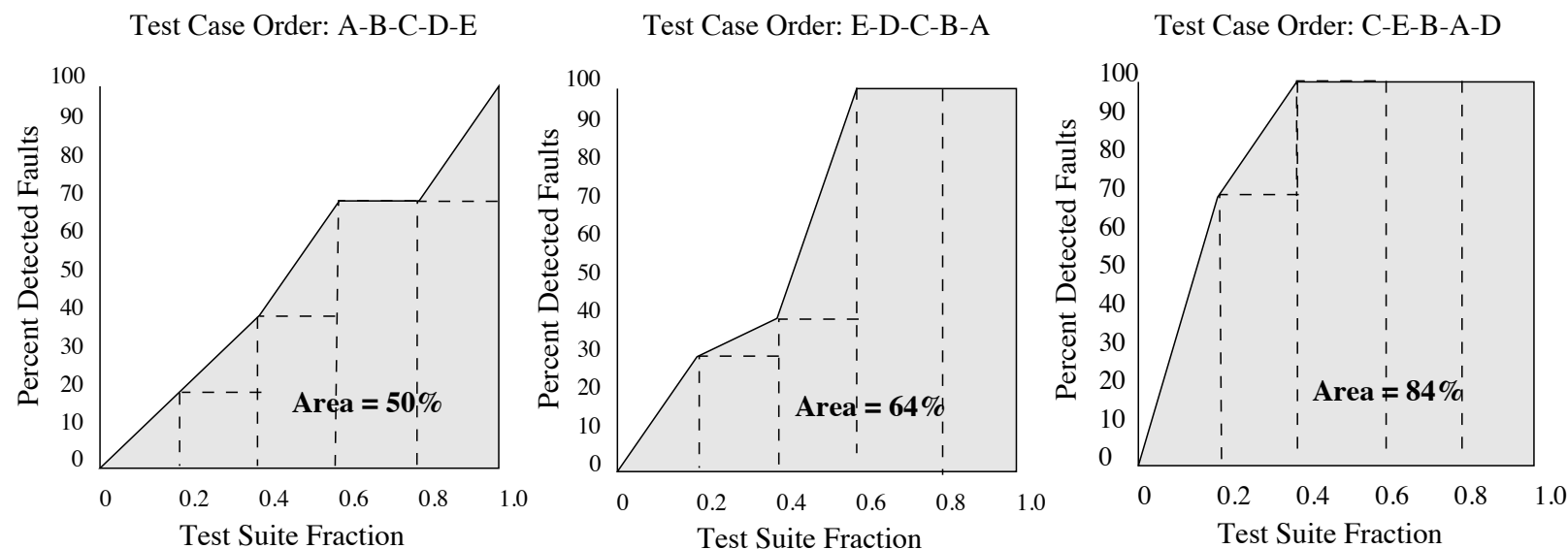
	s0	s1	s2	s3	s4	s5	s6	s7	s8	s9
t0	x				x					
t1	x				x	x	x			
t2	x	x	x	x	x	x	x			
t3					x					
t4								x	x	x

- It does not change things much; instead of trying to detect each fault, we try to cover each statement(branch/block/etc).
- The ideal ordering still begins with t2-t4.
- After t4, we reset the coverage counter and start again, which gives t1; similarly, t0 and t3.

Average Percentage of Fault Detection (APFD)

	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9
t0	x				x					
t1	x				x	x	x			
t2	x	x	x	x	x	x	x			
t3					x					
t4								x	x	x

- Measures how quickly your prioritisation detected the faults.

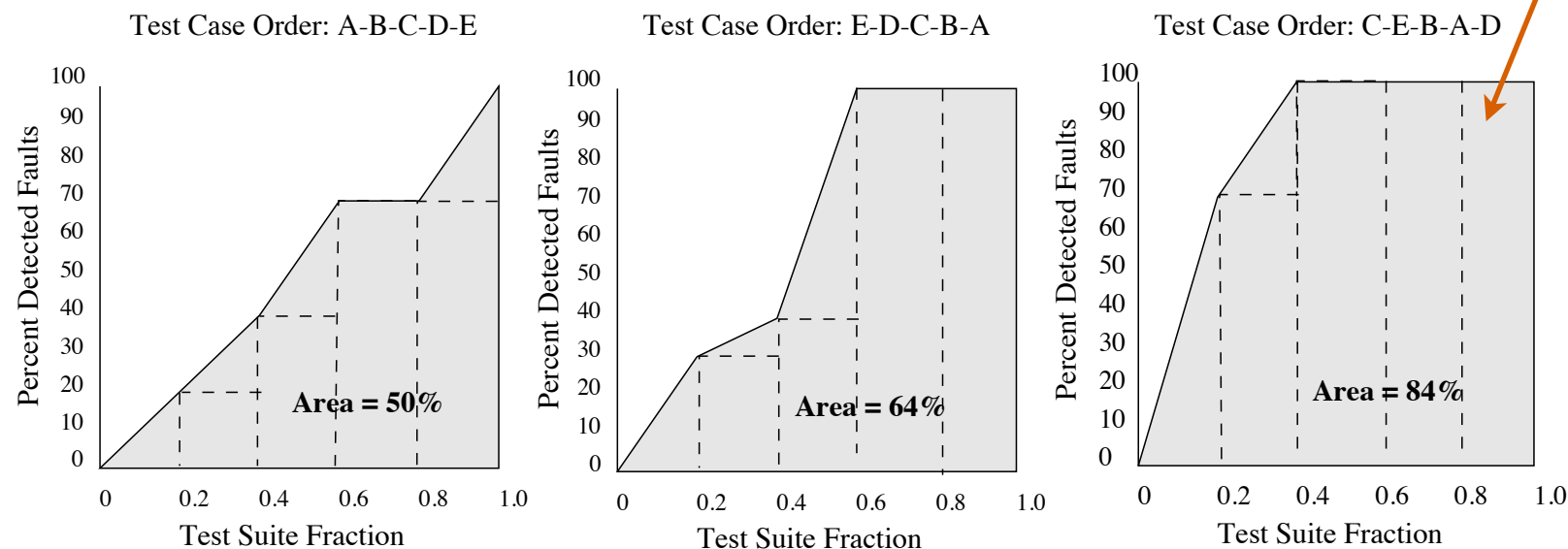


Average Percentage of Fault Detection (APFD)

	f0	f1	f2	f3	f4	f5	f6	f7	f8	f9
t0	x				x					
t1	x				x	x	x			
t2	x	x	x	x	x	x	x			
t3					x					
t4								x	x	x

Intuitively, the metric measures the area beneath.

- Measures how quickly your prioritisation detected the faults.

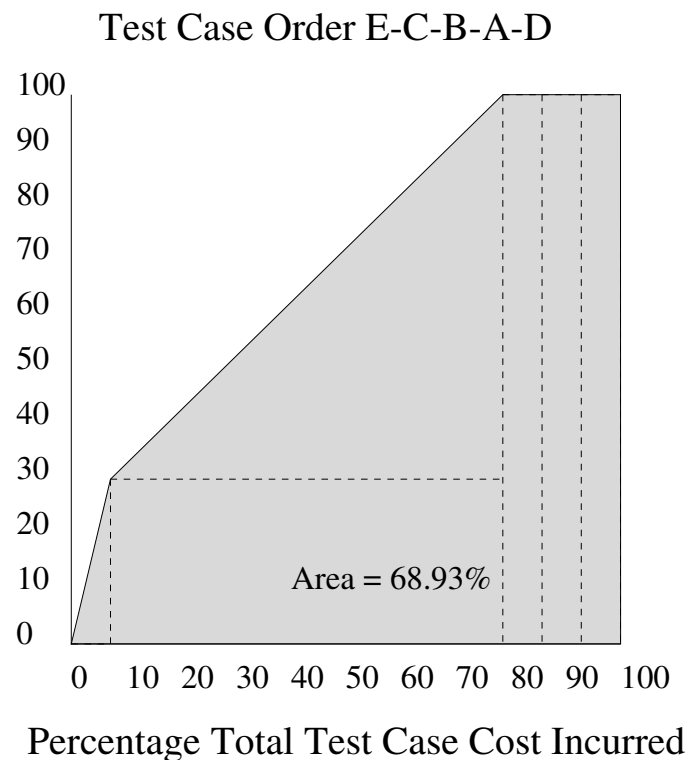
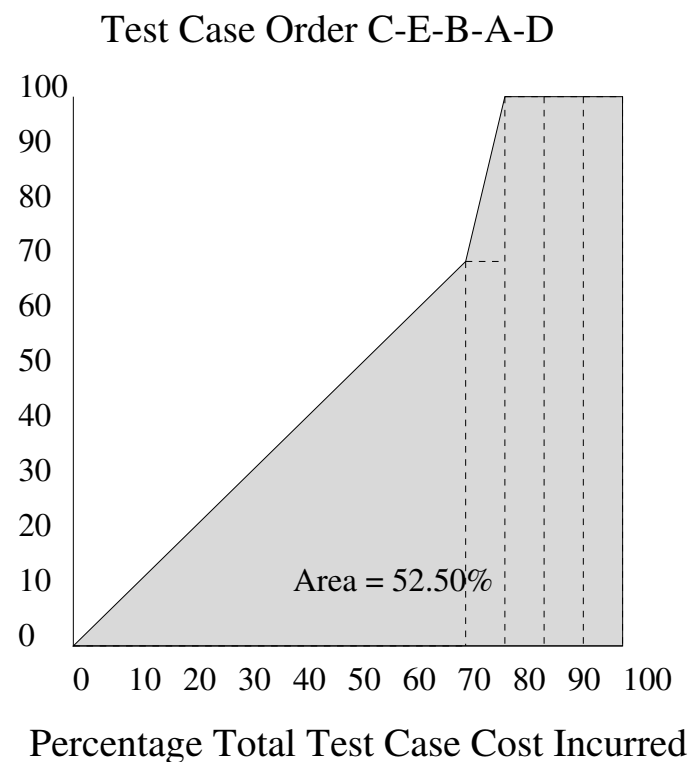


What about Cost?

- Yes, it is a recurring theme.
- Suppose test to achieves 100% code coverage in 5 hours.
Tests t1 to t3 collectively achieve 80% coverage in 2 hours.
Which one would you prioritise?

APFDc (Elbaum et al., 2001)

- C achieves 70% coverage in 7 minutes.
- E achieves 30% coverage in 30 seconds.
- As a result, executing E first gives higher APFDc.



Summary

- Minimisation: keyword is redundancy - it saves effort but you are putting yourself at the mercy of the criterion you are minimising with.
- Selection: keyword is safety - can be expensive; you want to be conservative and execute any test that has potential to reveal fault regarding recent changes.
- Prioritisation: keyword is asap/surrogate - you want to maximise the number of faults you detect early on; since you do not know faults in advance, you need to come up with smart surrogate.

Continuous Integration

- CI means to merge all developer working copies into a single mainline daily, or even more frequently.
- Developers usually ensure that their commits are correct by executing test cases that are directly relevant at their local machines. This is sometimes called pre-commit testing.
- Once changes are merged, the CI system automatically executes all relevant test cases, to ensure that individual changes correctly work with each other. This is sometimes called post-commit testing.

Regression Testing and Cloud Computing

- All regression testing techniques we looked at assume limited resource for testing, hence the need for optimisation.
- The elasticity of cloud computing has a significant implication on the way these techniques are used.
- What happens if you can use unrestricted amount of computing resources?

Single Testing Machine



Single Testing Machine



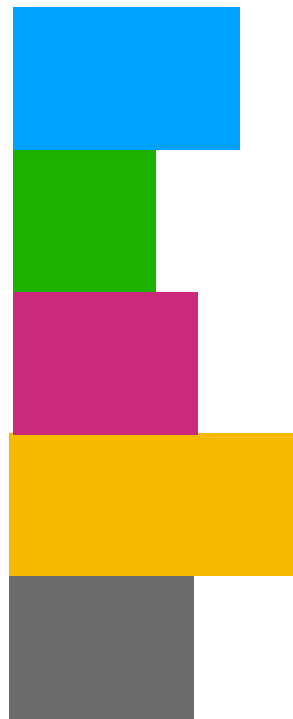
If all test cases have to be executed sequentially, we have to optimise the regression testing process as much as possible, to get the most out of the limited testing resources.

Single Testing Machine



If all test cases have to be executed sequentially, we have to optimise the regression testing process as much as possible, to get the most out of the limited testing resources.

Elastic Cloud Resource / Parallel Instances

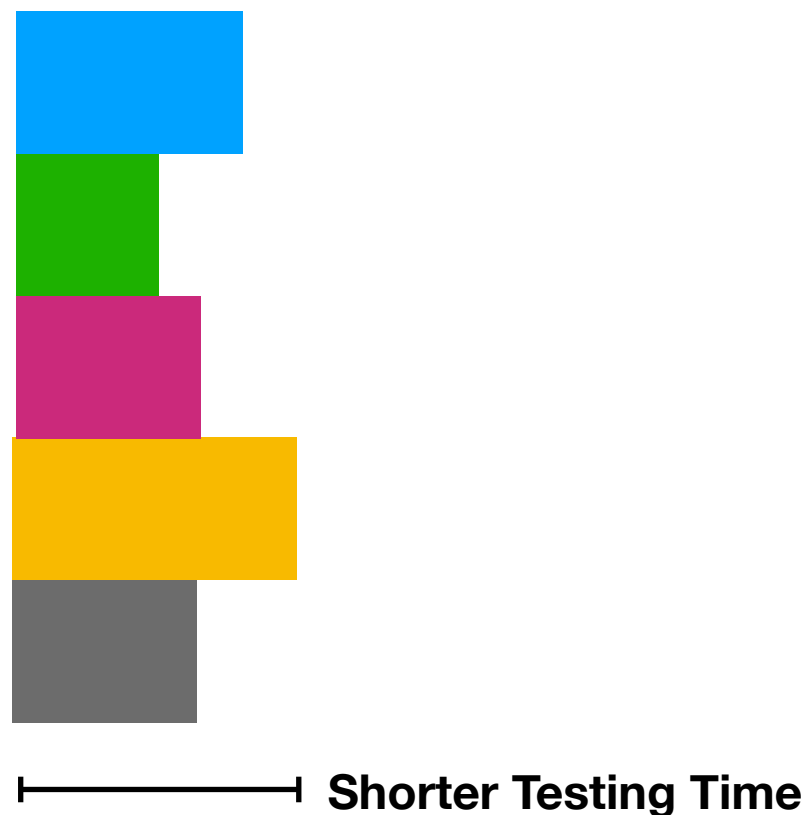


Single Testing Machine



If all test cases have to be executed sequentially, we have to optimise the regression testing process as much as possible, to get the most out of the limited testing resources.

Elastic Cloud Resource / Parallel Instances



If all test cases can be executed in parallel, the time needed for testing is only as long as the longest test execution.

So is it all useless?

- Many techniques are still meaningful even when test cases are executed in parallel.
 - Test suite minimisation can still save redundant cloud computing cost.
 - Impact analysis used by safe test case selection is useful for many other applications, such as automated debugging.
- There are cases when test cases have to be executed sequentially: for example, pre-commit testing done at individual developer machines.

Redefining Prioritisation

- In very large development environments, the CI (Continuous Integration) pipeline is easily **flooded** with commits.
 - Amazon engineers conduct 136,000 system deployments per day: one in every 12 seconds.
 - Google engineers have to wait up to 9 hours to receive test results from the CI pipeline.
- Prioritising test cases within test suites makes little impact at this scale.
- Instead, prioritising commits to test has been proposed.

Redefining Prioritisation

- The proposed technique is essentially history based prioritisation: commits relevant to test cases that have recently failed are given higher priority. If tests really fail, this ensures quicker feedback to the responsible developers.
- Assumptions
 - Commits are independent from each other. In high volume environment, this is not unrealistic.
 - Relationships between code and test cases are known in advance (i.e., which test covers which parts of the code).

Regression Testing and CI

- Modern software development requires increasingly shorter release cycles: time-to-market is extremely critical.
- Consequently, test executions are more frequently needed.
- While the CI environment requires adaptations, regression testing techniques are definitely relevant when test cases are repeatedly executed. In some sense, all testing is like regression testing.

Reference

- S. Yoo and M. Harman. Regression testing minimisation, selection and prioritisation: A survey. *Software Testing, Verification, and Reliability*, 22(2):67–120, March 2012.