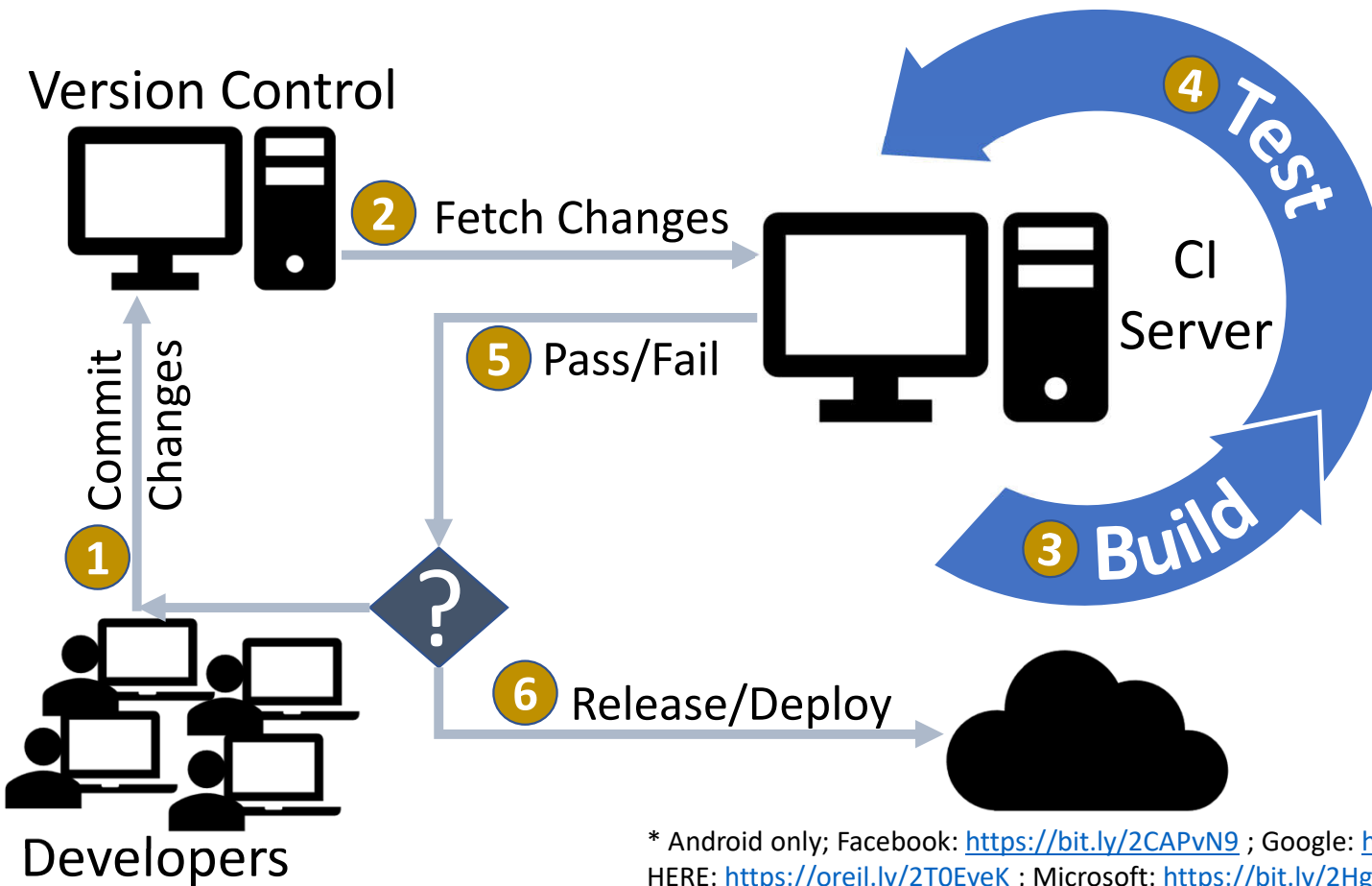


CS 5154
Regression Testing Techniques

Spring 2021

Owolabi Legunsen

Continuous Integration (CI): rapid test/release cycles



Builds per day:

- Facebook: 60K*
- Google: 17K
- HERE: 100K
- Microsoft: 30K
- Single open-source projects: up to 80

Releases per day

- Etsy: 50

* Android only; Facebook: <https://bit.ly/2CAPvN9> ; Google: <https://bit.ly/2SYY4rR> ;
HERE: <https://oreil.ly/2T0EyeK> ; Microsoft: <https://bit.ly/2HgjUpw> ; Etsy: <https://bit.ly/2liSOJP> ;

Several important problems exist in these cycles

P1: Passing tests miss bugs

S1: Find more bugs from tests that developers already have

P3. Testing can be very slow

S3: Find bugs faster by speeding up testing

P2. Failed tests, no buggy changes

S2: Find bugs more reliably by detecting such failures

P4. How to test in new domains?

S4: Find bugs in emerging application domains

The problem that we'll talk about today








Problem: Testing can be very slow

Solution: Techniques that can help speed up regression testing



* Android only; Facebook: <https://bit.ly/2CAPvN9> ; Google: <https://bit.ly/2SYY4rR> ;
HERE: <https://oreil.ly/2T0EyeK> ; Microsoft: <https://bit.ly/2HgJUpw> ; Etsy: <https://bit.ly/2liSOJP> ;

Re-running tests can be very slow

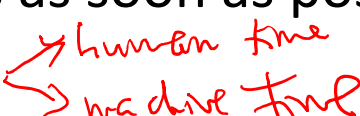



	test execution time	number of tests
	~5min	1667
 guava-libraries <small>Guava: Google Core Libraries for Java 1.6+</small>	~10min	641534
	~45min	1296
	~45min	361
	~45min	631
	~4h	4975
	~17h	8663

Run many times each day

What are your ideas for speeding up testing?

- parallelize
- run test affected by changes \sqrt{RTS}
- write fewer tests that satisfy stronger criteria
- pick and choose test based how long they take and ~~long~~ ^{budget} prioritization
- run test outside work hours
- designate core tests and random fraction of others ✓

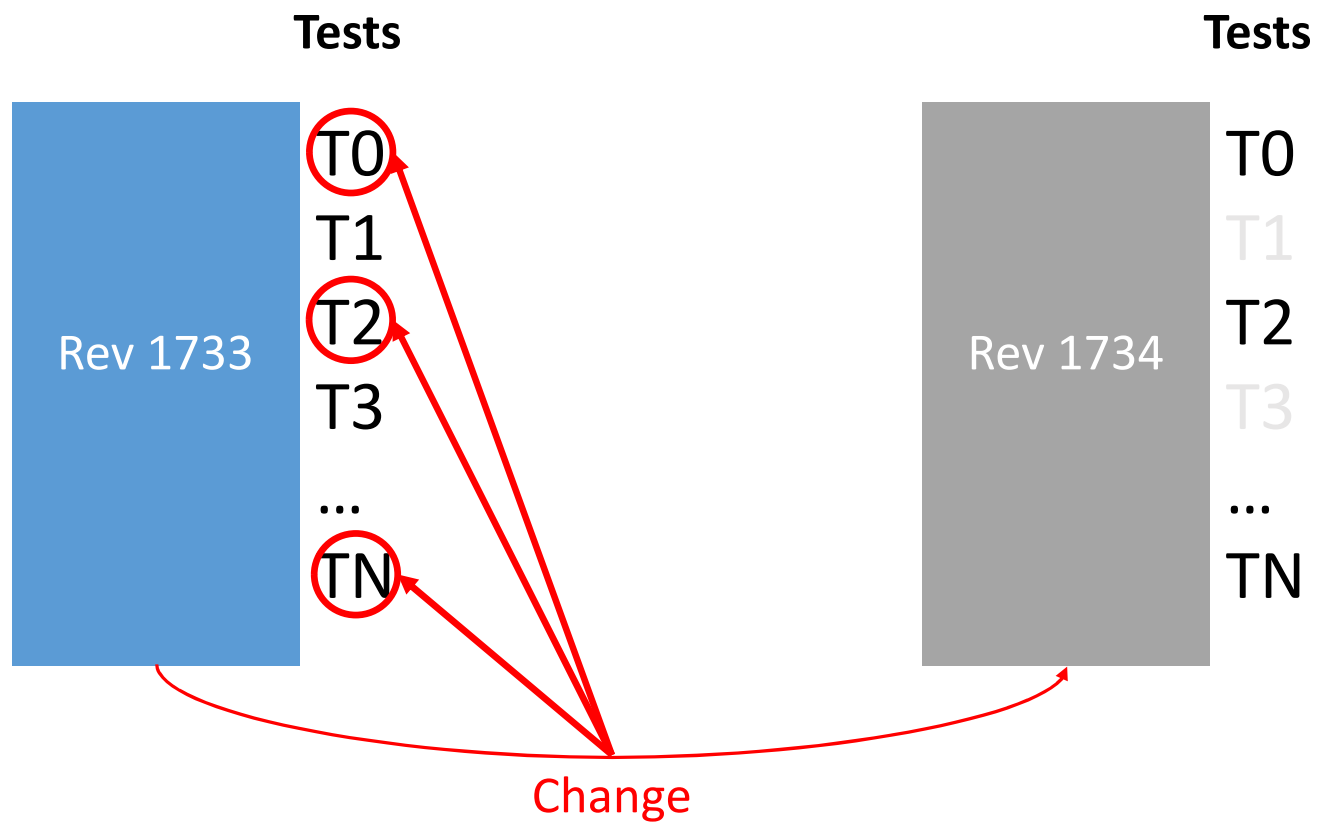
In this lecture

- Speed up regression testing
 - Detect regression faults as soon as possible
 - Reduce cost of testing 
- Common techniques:
 - Regression Test Selection 
 - Test-Suite Reduction (Minimization) 
 - Test-Case Prioritization 

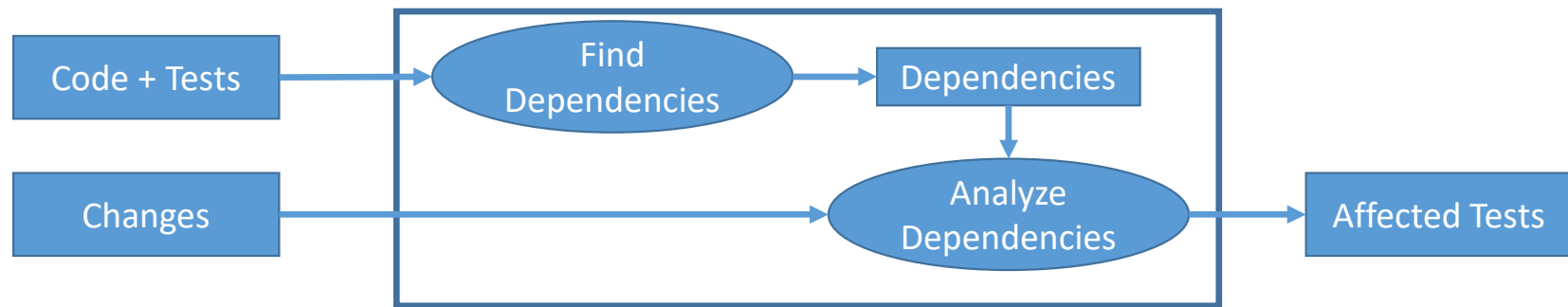
Regression Testing Techniques

- Speed up regression testing
 - Detect regression faults as soon as possible
 - Reduce cost of testing
- Common techniques:
 - **Regression Test Selection**
 - Test-Suite Reduction (Minimization)
 - Test-Case Prioritization

Regression Test Selection (RTS)



How RTS works



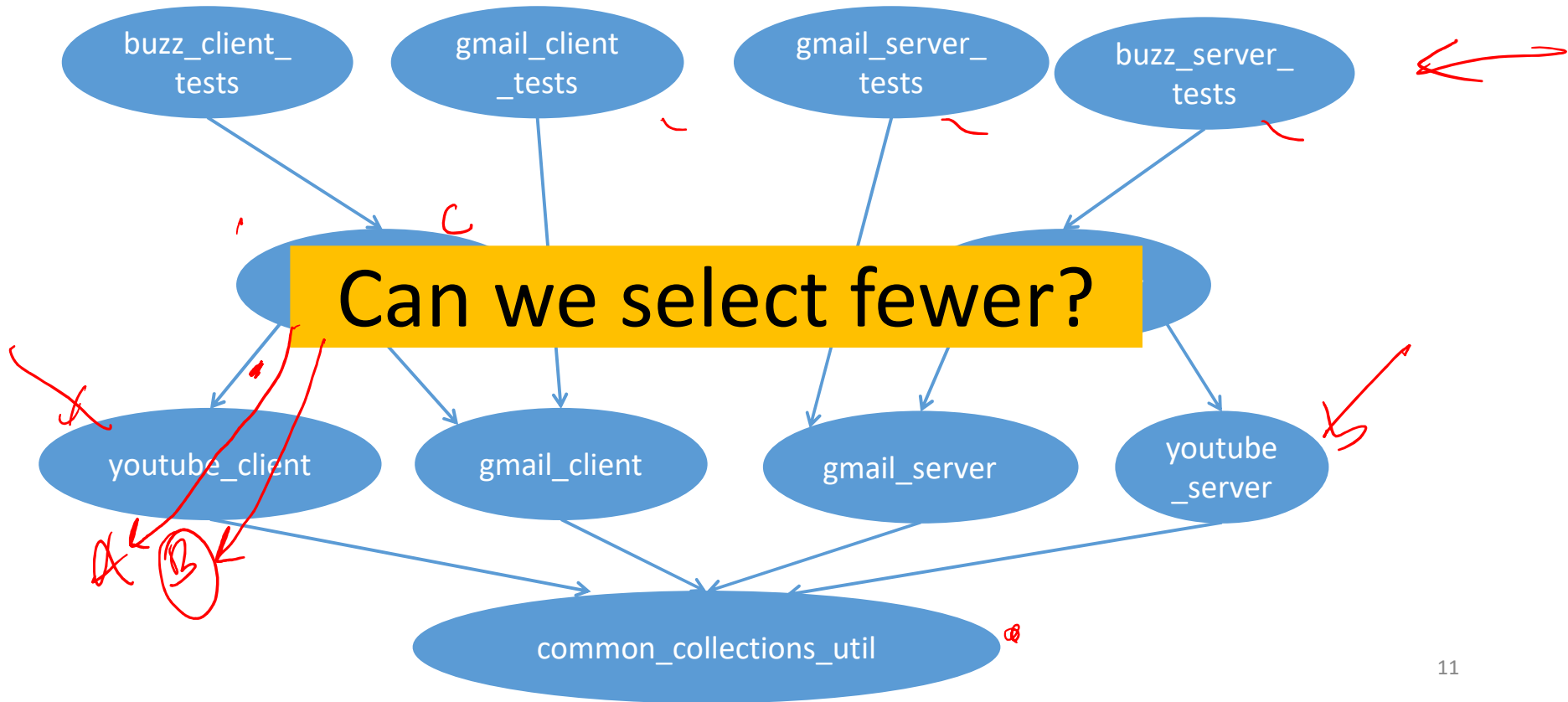
- An **affected test** can behave differently after?? due to code changes
- A test is affected if any of its dependencies changed

4

TAP

RTS at Google (Target/Module Level)

Bazel / Blaze



Class-level RTS

- Track dependencies between classes (in Java)
 - Collect changes at class level
 - Connect relationships between classes
 - Select test classes (run all test methods in selected test class)
- How do we track test dependencies?
- How do we track changes?



Ekstazi

Class-level Dynamic RTS (Ekstazi¹)

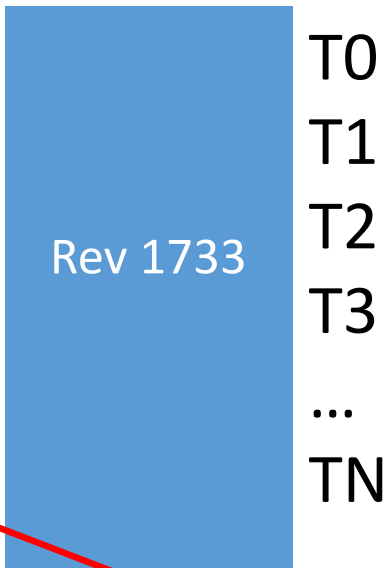
- **Find Dependencies:** dynamically track classes used while running each test class
 - Instrument classes to figure out which classes are used/loaded when running tests in some test class
- **Changes:** classes whose .class (bytecode) files differ
- **Analyze Dependencies:** select test classes for which any of its dependencies changed
 - Maintain dependencies between versions

source ?!

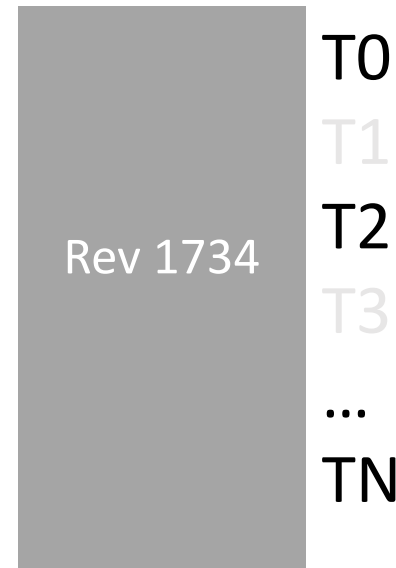
¹Gligoric et al., *Practical Regression Test Selection with Dynamic File Dependencies*. ISSTA 2015, <https://github.com/gliga/ekstazi>

Ekstazi Example

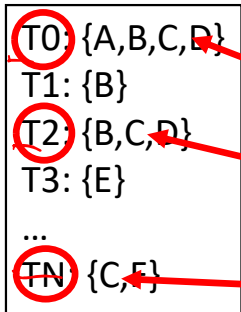
Tests



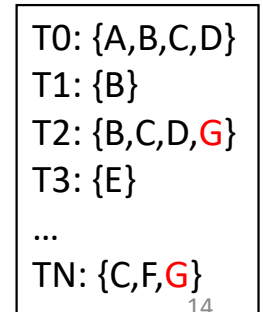
Tests



Ekstazi Dependencies



Ekstazi Dependencies



Change

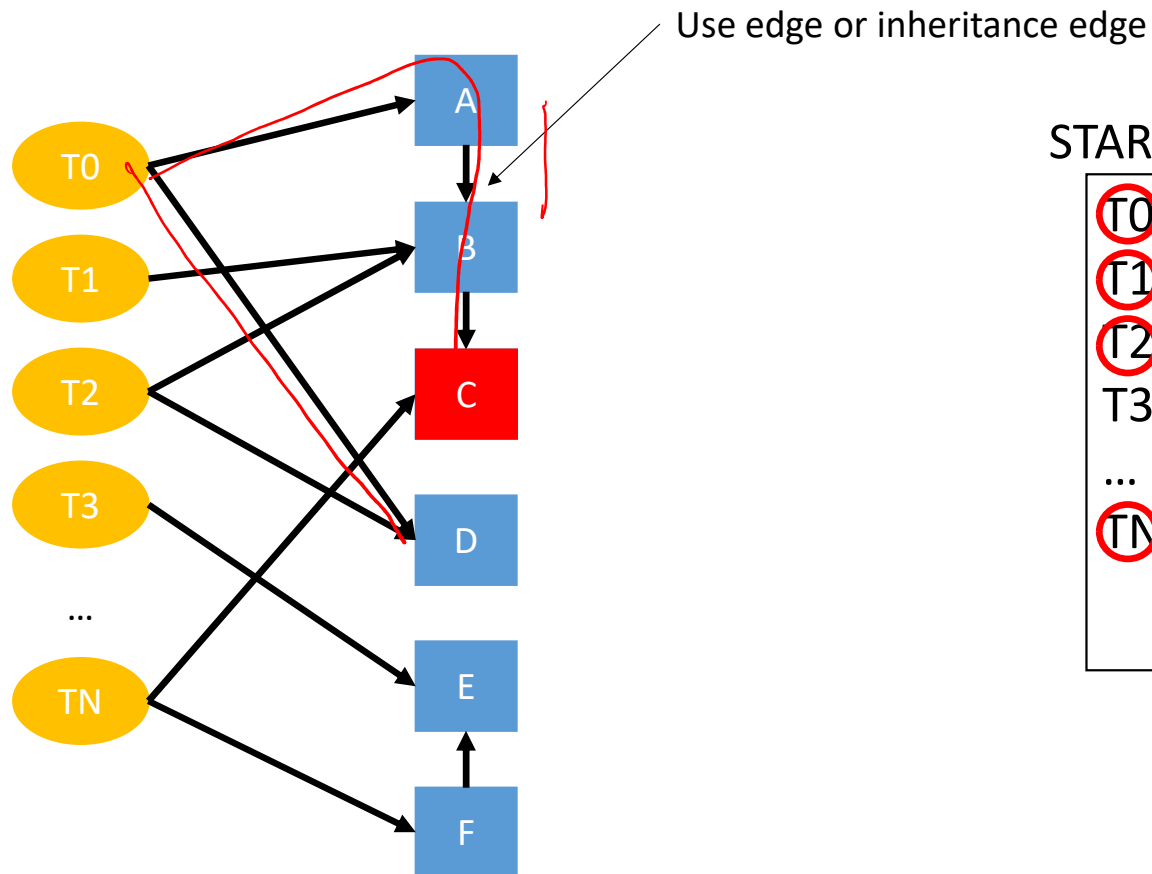
{C} T_{n+1}

Class-level STatic RTS (STARTS¹)

- First, statically build a class dependency graph
 - Each class has an edge to direct superclass/interface and referenced classes
- **Find Dependencies:** classes reachable from test class in the graph
- **Changes:** computed in same way as Ekstazi
- **Analyze Dependencies:** select test classes that reach a changed class in the graph

¹Legunsen et al., *An Extensive Study of Static Regression Test Selection in Modern Software Evolution*. FSE 2016

STARTS Example

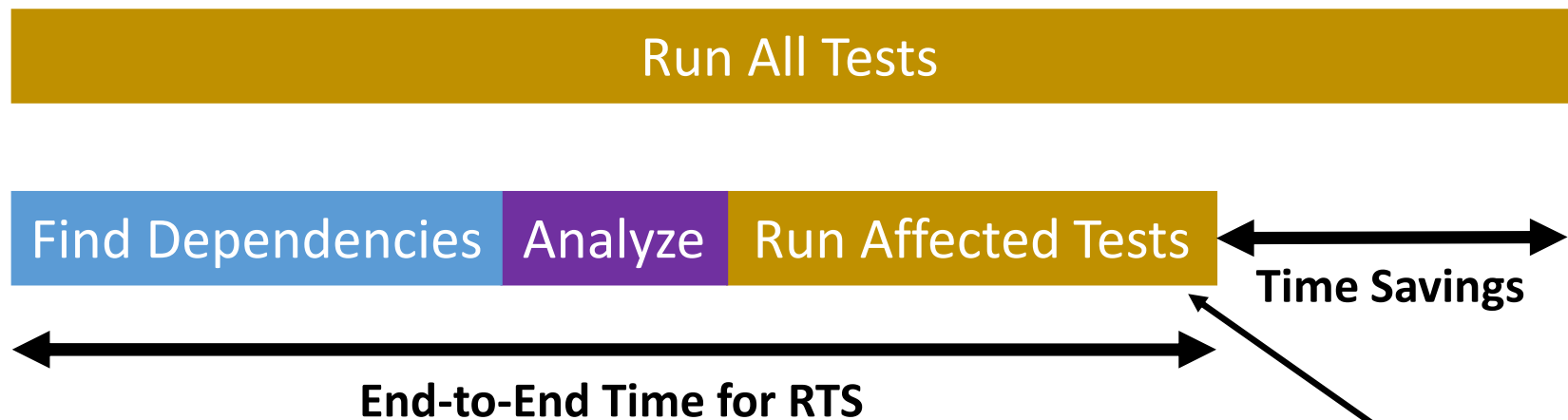


STARTS Dependencies

- $\Gamma_0: \{A, B, C, D\}$
- $\Gamma_1: \{B, C\}$
- $\Gamma_2: \{B, C, D\}$
- $T_3: \{E\}$
- ...
- $\Gamma_N: \{C, E, F\}$

Transitive closure

Important RTS Considerations



- RTS is **safe** if it selects to rerun *all* affected tests

For Ekstazi, includes time to run and collect coverage/dependencies

- RTS is **precise** if it selects to rerun *only* affected tests ✓

Pros and cons of static vs. dynamic RTS?

dynamic may select fewer tests (pro)
static builds dependencies faster
static selects more tests

Dynamic vs Static

- Dynamic:

- Pro

- Gets exactly what tests depends on

- Con

- Requires executing tests to collect dependencies (overhead)

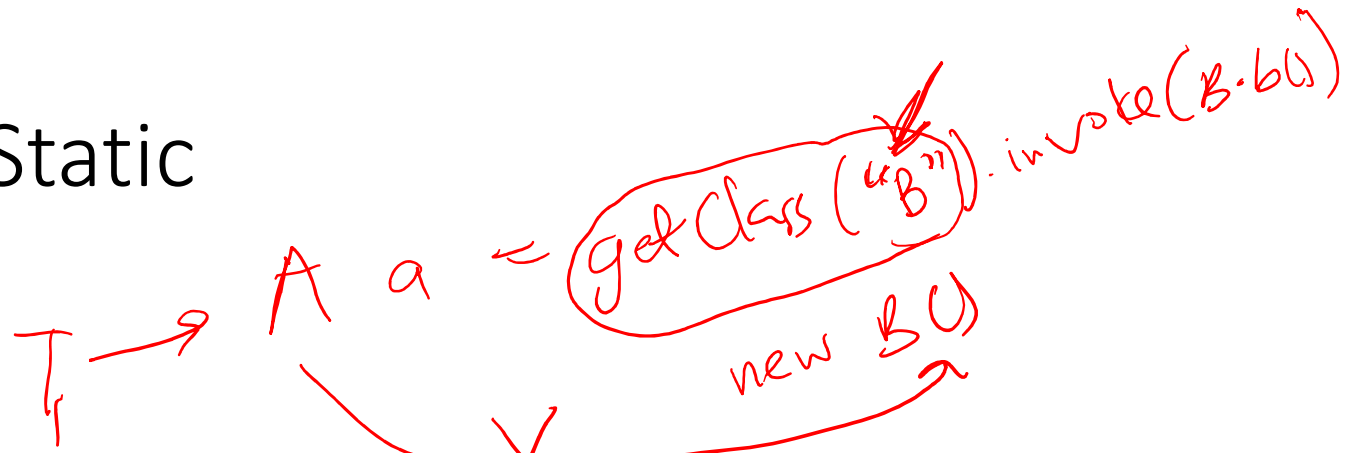
- Static:

- Pro

- Quick analysis without needing to execute tests

- Con

- Can over-approximate affected tests due to static analysis
- May miss dependencies (reflection!)



Finer Granularity?

- Why not go even finer granularity of dependencies?
 - Method-level?
 - Statement-level?
- Collecting such dependencies (correctly) is harder *Costlier*
- More time to collect dependencies
 - Is the extra time worth it?
- Can actually be unsafe!

Class-level vs Target/Module-level

- Class-level test selection should be more precise than target/module-level test selection
 - Selects to run all tests in affected test class, not all tests in affected test target/module
- Why do companies not use class-level test selection?

— inertia??

Some RTS tools you can use today

- Built by researchers (click on links below)

- [STARTS](#) ✓
- [Ekstazi](#) ✓

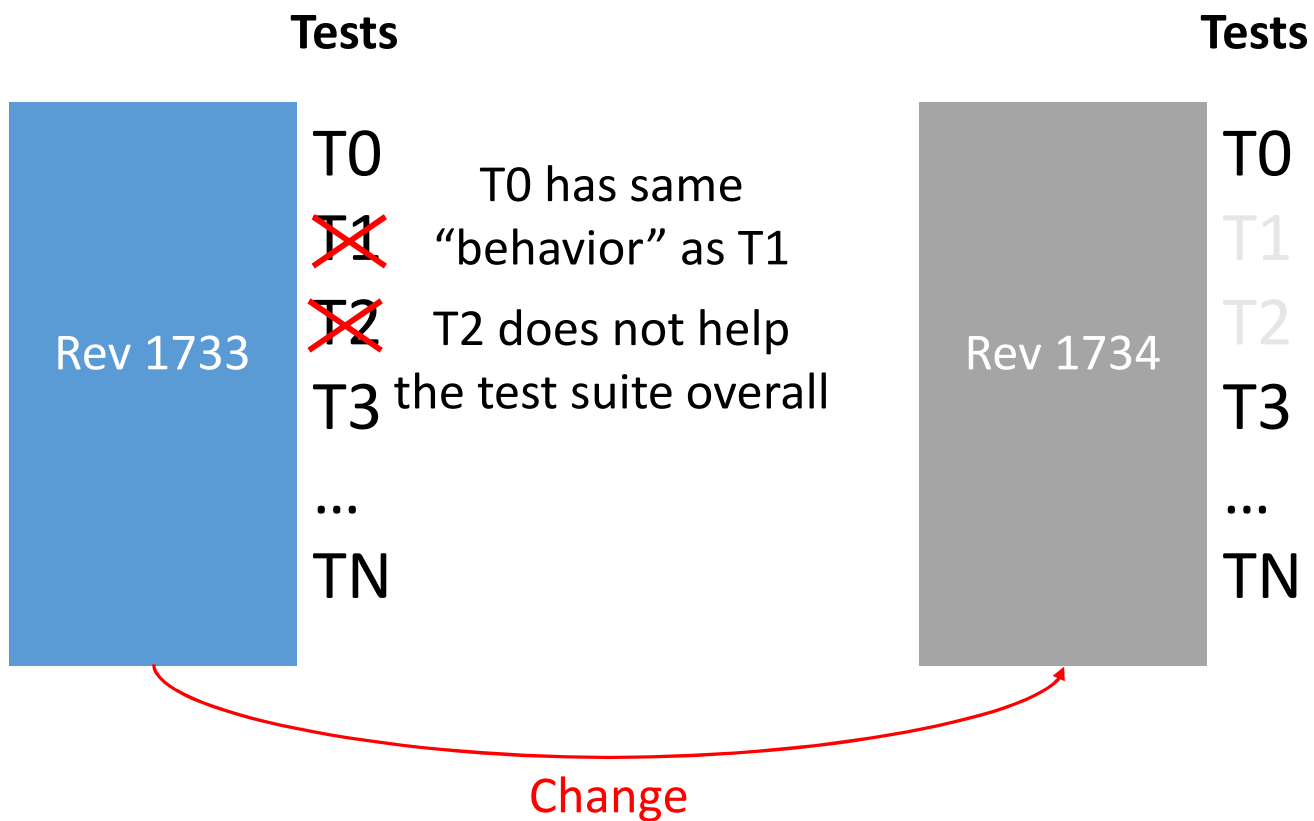
- Built by industry (click on links below)

- [Microsoft Test Impact Analysis](#) ♦
- [OpenClover Test Optimization](#)

Regression Testing Techniques

- Speed up regression testing
 - Detect regression faults as soon as possible
 - Reduce cost of testing
- Common techniques:
 - Regression Test Selection
 - **Test-Suite Reduction (Minimization)**
 - Test-Case Prioritization

Test-Suite Reduction (TSR)



Test-Suite Reduction (TSR)

- Create a smaller, reduced test suite to run
 - Run fewer tests overall across many revisions
- Analysis happens once/infrequently, so okay to spend more time
- Test suite should not miss to detect any faults
 - Ideal: all tests that would fail and detect fault should be kept in the reduced test suite
 - Just as good: at least one test that can detect each fault should be kept in the reduced test suite

TSR versus RTS

	Test-Suite Reduction	Regression Test Selection
How are tests chosen to run?	Redundancy (one revision)	Changes (two revisions)
How often is analysis performed?	Infrequently	Every revision
Can it miss failing tests from the original test suite?	Yes	No (if safe)

TSR Process

Heuristic: tests that cover the same elements as the original test suite are just as good

T = Tests
 C = Classes
 (could be statements, methods, branches, etc.)
 M = Mutants
 (could be other fault-like requirements)

	C0	C1	C2	C3	C4	M1	M2	M3	M4
T0	X	X				X			
T1		X	X					X	X
T2		X	X					X	X
T3			X				X	X	X
T4	X				X	X			

Reduced Test Suite R = {T2,T4}

Size

$$Siz = \frac{|R|}{|O|} = 40\%$$

Fault-Detection Capability

$$ReqLoss = \frac{|req(O) \setminus req(R)|}{|req(O)|} = 25\%$$

Req ∈ {class, mutant}

TSR Algorithms

- TSR is essentially set cover problem (NP-Complete)
- Algorithms to approximate finding minimal test suite:
 - Greedy (Total vs Additional)
 - GRE
 - GE
 - HGS *→ Authors' initials*
 - ILP (Integer Linear Programming -> can get actual minimal)

Greedy Algorithm (Total)

- Greedy heuristic: select test that covers the most elements
- Iteratively make greedy choice test
 - Tie-break: Random? Sorted by name?
- Stop when chosen tests cover all elements

Greedy (Total) Example

	C0	C1	C2	C3	C4
T0	X	X			
T1		X	X		
T2		X	X		
T3			X		
T4	X				X

req(O) = {~~C0~~, ~~C1~~, C2, C4}

R = {T0}

Greedy (Total) Example

	C0	C1	C2	C3	C4
→ T0	X	X			
→ T1		X	X		
T2		X	X		
T3			X		
T4	X				X

req(O) = {~~C0~~, ~~C1~~, ~~C2~~, C4}

R = {T0, T1}

Greedy (Total) Example

	C0	C1	C2	C3	C4
→ T0	X	X			
→ T1		X	X		
→ T2		X	X		
T3			X		
T4	X				X

req(O) = {~~C0~~, ~~C1~~, ~~C2~~, C4}

R = {T0, T1, T2}

Greedy (Total) Example

	C0	C1	C2	C3	C4
→ T0	X	X			
→ T1		X	X		
→ T2		X	X		
T3			X		
→ T4	X				X

req(O) = {~~C0~~, ~~C1~~, ~~C2~~, ~~C4~~}

R = {T0, T1, T2, T4}

Greedy Algorithm (Additional)

- Greedy heuristic: select test that covers the most *uncovered* elements
 - Keep track of what has been covered so far and only consider the yet-to-be covered ones
- The rest is the same as Greedy (Total)

Greedy (Additional) Example

	C0	C1	C2	C3	C4
→ T0	X	X			
T1		X	X		
T2		X	X		
T3			X		
T4	X				X

req(O) = {~~C0~~, ~~C1~~, C2, C4}

R = {T0}

Greedy (Additional) Example

	C0	C1	C2	C3	C4
→ T0	X	X			
T1		X	X		
T2		X	X		
→ T3			X		
T4	X				X

req(O) = {~~C0~~, ~~C1~~, ~~C2~~, C4}

R = {T0, T3}

Greedy (Additional) Example

	C0	C1	C2	C3	C4
→ T0	X	X			
T1		X	X		
T2		X	X		
→ T3			X		
→ T4	X				X

req(O) = {~~C0~~, ~~C1~~, ~~C2~~, ~~C4~~}

R = {T0, T3, T4}

GRE ↙

cover

- Iteratively select essential tests
 - An essential test uniquely covers some elements that no other test can cover
 - Select the “most” essential tests that cover most unique elements
- Selecting some essential tests may make other tests essential
 - If no essential tests, then make the greedy (additional) choice

GRE Example

	C0	C1	C2	C3	C4
T0	X	X			
T1		X	X		
T2		X	X		
T3			X		
→ T4	X				X

req(O) = {~~C0~~, C1, C2, ~~C4~~}

R = {T4}

GRE Example

	C0	C1	C2	C3	C4
T0	X	X			
T1		X	X		
→ T2		X	X		
T3			X		
→ T4	X				X

req(O) = {~~C0~~, ~~C1~~, ~~C2~~, ~~C4~~}

R = {T4, T2}

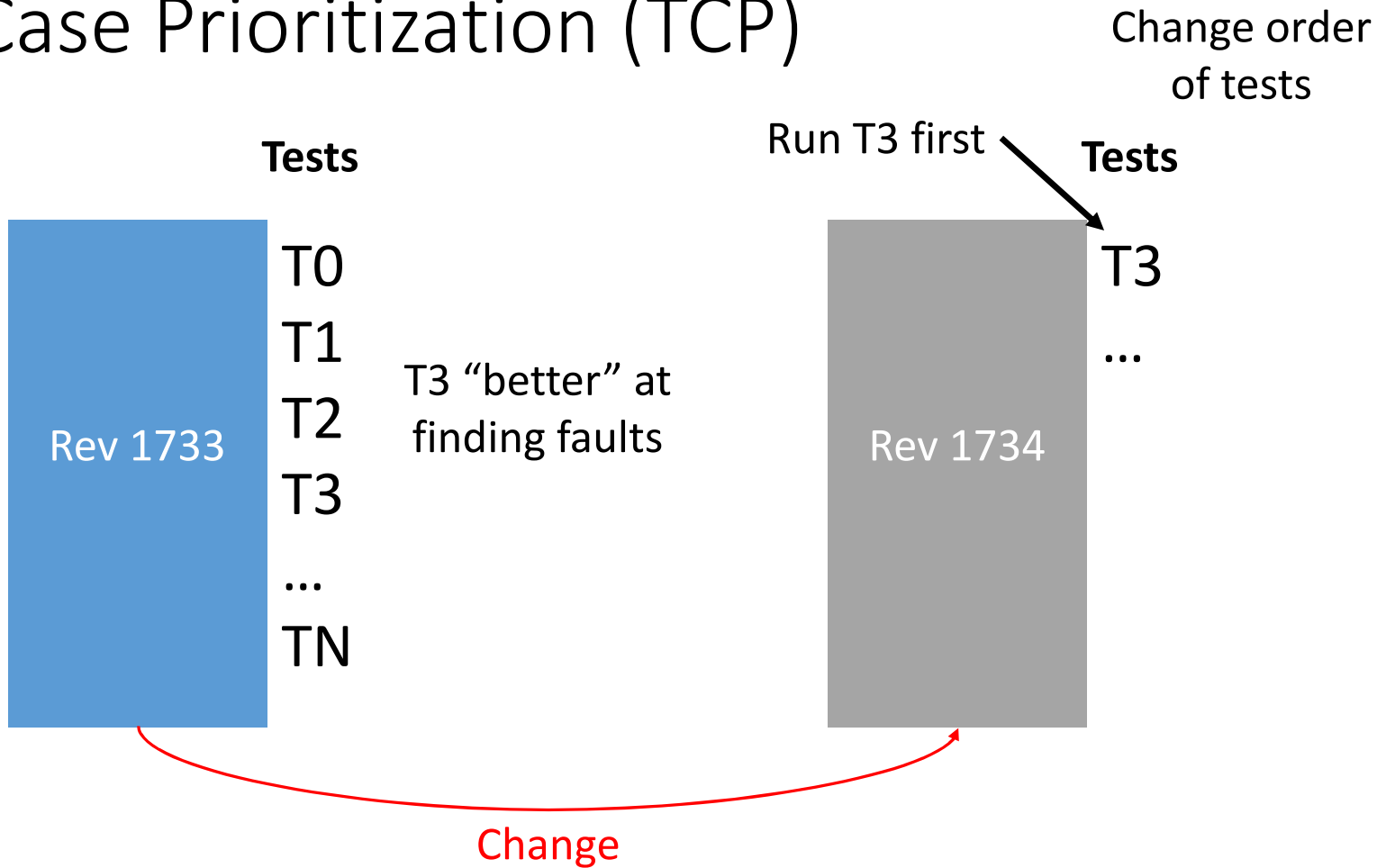
Open Questions

- Can reduced test suite replace original test suite for future revisions?
 - Can reduced test suite fail when original test suite does?
- Are removed tests truly redundant?
 - Would you trust the algorithm in removing some of your tests?
 - What heuristics should we use to determine redundancy?
- Does evaluation with seeded faults/mutants on current version predict effectiveness in future?

Regression Testing Techniques

- Speed up regression testing
 - Detect regression faults as soon as possible
 - Reduce cost of testing
- Common techniques:
 - Regression Test Selection
 - Test-Suite Reduction (Minimization)
 - **Test-Case Prioritization**

Test-Case Prioritization (TCP)



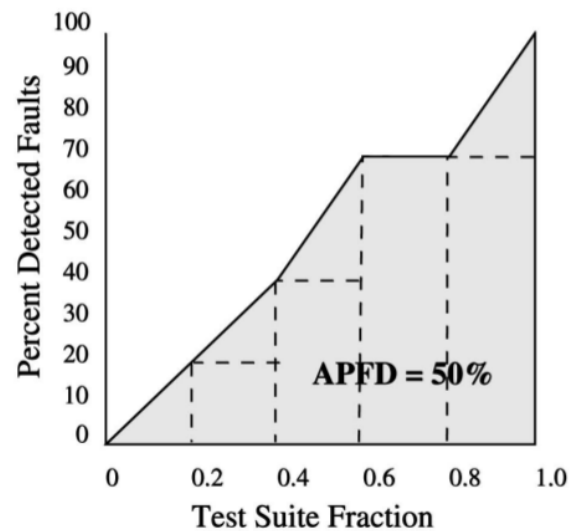
Test-Case Prioritization (TCP)

- Run all tests, but in decreasing order of likelihood of revealing faults
 - As tests run, debug and fix faults discovered by early test failures
- Runs all tests, so
 - no risk of missing any test failure
 - overall cost does not go down
- “Poor person’s” test selection: stop running when budget is exceeded

Microsoft's Echelon (circa 2002)

How to Evaluate TCP Orders?

- Similar to TSR, evaluate using other requirements, e.g., faults/mutants
- Measure “speed” of detecting all faults



APFD

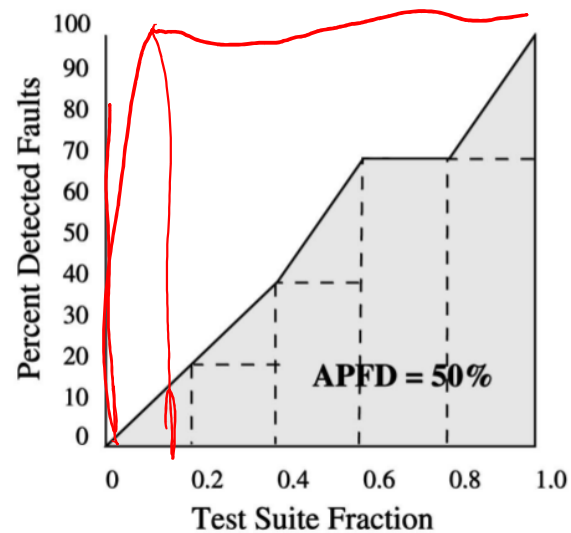
- **Average Percentage of Faults Detected (APFD)**

$$1 - \frac{\sum_{i=1}^m TF_i}{n \times m} + \frac{1}{2n}$$

n = number of tests

m = number of faults

TF_i = position of test that detects fault i



TCP Algorithms

- Prioritize based on coverage
 - Greedy (Total vs Additional) }
 - Adaptive Random }
- Prioritize based on source code
 - Order tests based on source code differences
 - Information retrieval ✓
- Simple (yet effective!) prioritization
 - Quickest test first
 - Most frequently failing (historically) test first

Greedy TCP

- Similar to previous Greedy algorithms for TSR
 - Greedy choice: test that covers the most (uncovered) elements
- Eventually, all tests still get run

Adaptive Random TCP

- Start with a random test
- Order next tests based on greatest “dissimilarity” with prioritized tests
 - E.g., for coverage, which tests cover the most different elements than any of the tests already prioritized?
 - Measure distance between covered elements (e.g., Jaccard distance)
 - Maximum distance? Minimum distance?

Adaptive Random TCP Example

$$D_{jaccard} = 1 - \frac{|A \cap B|}{|A \cup B|}$$

$$1 - \frac{1}{3} = \frac{2}{3}$$

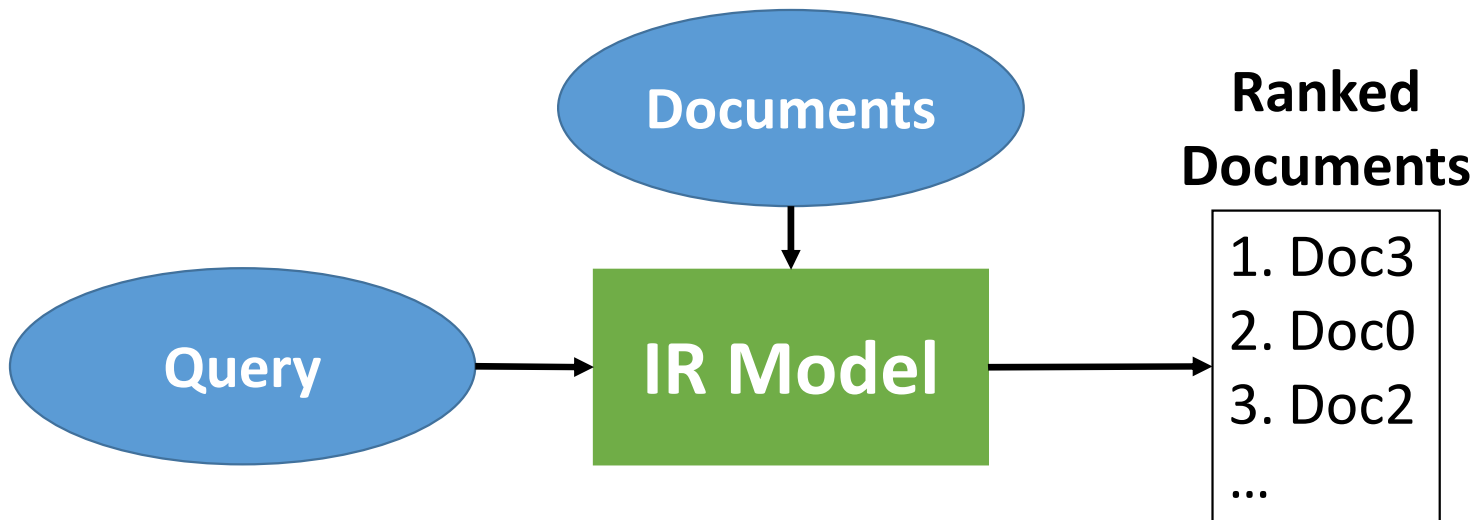
	C0	C1	C2	C3	C4			
→ T0	X	X				}	2/3	2/3
→ T1		X	X				0	0
T2		X	X			1/2	1/2	1/2
T3			X					
→ T4	X				X	1		

[T1,T4,T0,T3,T2]

Information Retrieval

Information Retrieval (IR)

Rank text documents based on relevance to a query

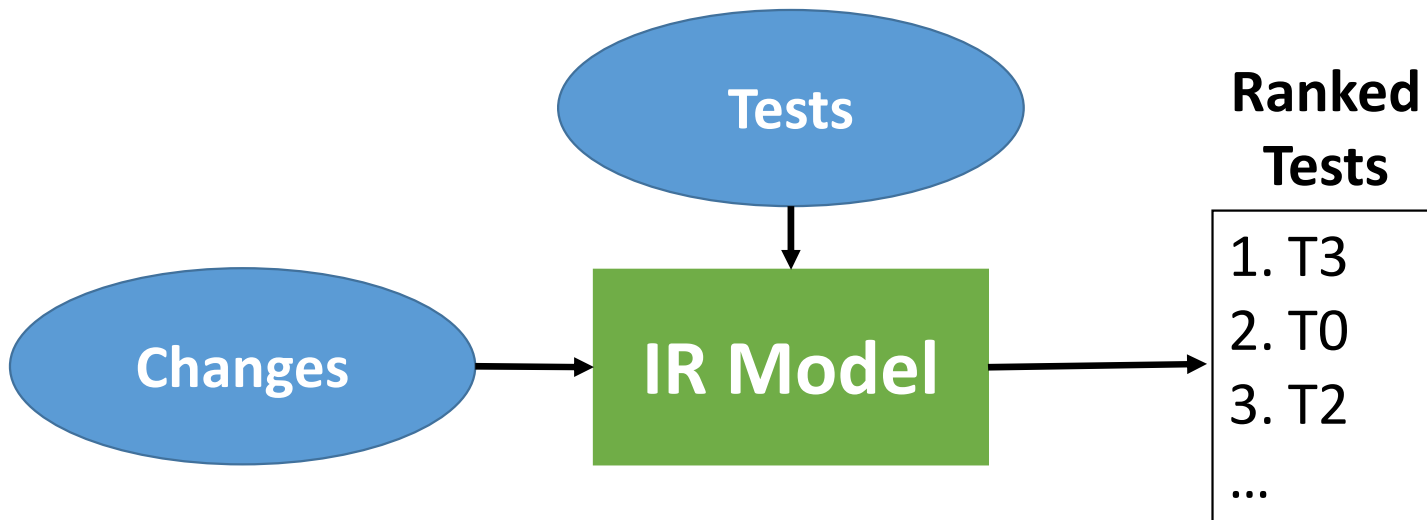


Information Retrieval TCP

Information Retrieval (IR)-Based TCP¹

Rank tests based on relevance to changed code

Change-aware



¹Saha et al., "An information retrieval approach for regression test prioritization based on program changes", ICSE 2015

Open Questions

- How to best model likelihood of test failing?
 - Is coverage a good heuristic?
 - Is “diversity” what we want?
 - Can it be done quickly? Change-aware?
- Should TCP be used in companies?
 - Does it make sense to prioritize tests in order of likely failing? ✓
 - Does it make sense to have mindset of debugging as soon as test fails, even as tests keep running?

Summary

- Regression testing can be a huge cost of software development
- Regression testing techniques aim to reduce that cost
- Many other techniques exist for reducing the cost
 - Test parallelization
 - Test mocking
 - Test slicing
 - Optimizing test placement
 - Machine learning approach for RTS, TCP, etc.