Class 16

- Questions/comments
- Graders for Problem Set 6 (4); Graders for Problem set 7 (2) (solutions for all)
- Testing, regression testing
- Assign (see Schedule for links)
  - Problem Set 6 discuss
  - Readings

Subsumption Hierarchy

Frankl and Weyuker presented a hierarchy of some criteria that they discussed in their paper

Show their relationship among the following criteria

- All paths
- All du-paths
- All uses
- All defs
- All branches
- All nodes

Data-Flow Coverage Criteria: Review

- Most popular criteria
  - All uses
  - All du-paths
- Give an example that shows how they differ in the test requirements and test cases

Mutation Analysis/Testing

- Basic idea: Generate a set of programs Π similar to the program P (mutants) under test and run the test suite T on P and on all programs in Π
- Differentiating (killing) programs:

  A test case differentiates two programs if it causes the two programs to produce different results
- Selection criteria: T is selected so that for each program P' in Π there exists at least a t in T that differentiates P from P'
- Evaluation criteria: The quality of T is related to the ability of T to differentiate P from programs in Π

Mutation Analysis/Testing

- Based on how Π is generated (P' more or less similar to P), we can perform analysis at different levels of detail
- The main problem is the generation of mutants
  - Ideal situation: one mutant for each possible fault in the program (obviously impractical)
  - Instead, we limit the cardinality of Π based on:
    - Application type
    - Types of faults that are more likely to occur
    - Programming language
- The main advantage is that the technique can be easily automated

Mutation Analysis/Testing

- A mutant operator is a function that, given P, generates one or more mutants of P
- The simplest operators perform simple syntactic modification to the code that result in semantic changes. There are different classes of operators:
  - Operators that work on constants, scalar variables, and arrays by replacing each occurrence of a variable with all other variables in scope
  - Operators that modify the operators in the program (e.g., ">" with "<"
  - Operators that replace expressions in the program with different expressions (e.g., constants)
  - Operator that modify the instructions in the program (e.g., a "while" transformed in an "if")
  - …
- The tester decides which operators to use and how many mutants to generate with the selected operators

Mutation Analysis/Testing: Example

```
1.  read i
2.  read j
3.  sum = 0
4.  while (i > 0) and (i < = 10)
      do
5.      if  (j >0)
6.          sum = sum + j
6a.         print sum
        endif
7.      i= i + 1
8.      read j
    endwhile
9. print  sum
```

Mutate: make a small syntactic change

Mutation Analysis/Testing: Example

```
1.  read i
2.  read j
3.  sum = 0
4.  while (i > 0) and (i < = 10)
      do
5.      if  (j >0)
6.          sum = sum + j
6a.         print sum
        endif
7.      i= i + 1
8.      read j
    endwhile
9. print  sum
```

Mutate: make a small syntactic change

Mutation: the changed statement

Mutation Analysis/Testing: Example

```
1.  read i
2.  read j
3.  sum = 0
4.  while (i > 0) and (i < = 10)
      do
5.      if  (j >0)
6.          sum = sum + j
6a.         print sum
        endif
7.      i= i + 1
8.      read j
    endwhile
9. print  sum
```

Mutate: make a small syntactic change

Mutation: the changed statement

Mutant: program with a mutated statement

```
1.  read i
2.  read j
3.  sum = 0
4.  while (i > 0) and (i < = 10)
      do
5.      if  (j >0)
6.          sum = sum - j
6a.         print sum
        endif
7.      i= i + 1
8.      read j
    endwhile
9. print  sum
```

Mutation Analysis/Testing: Example

```
1.  read i
2.  read j
3.  sum = 0
4.  while (i > 0) and (i < = 10)
      do
5.      if  (j >0)
6.          sum = sum + j
6a.         print sum
        endif
7.      i= i + 1
8.      read j
    endwhile
9. print  sum
```

Mutate: make a small syntactic change

Mutation: the changed statement

Mutant: program with a mutated statement

```
1.  read i
2.  read j
3.  sum = 0
4.  while (i > 0) and (i  <  10)
      do
5.      if  (j >0)
6.          sum = sum + i
6a.         print sum
        endif
7.      i= i + 1
8.      read j
    endwhile
9. print  sum
```

Mutation Analysis/Testing: Systems

- Mothra Mutation System for Fortran
  - Jeff Offutt and Rich DeMillo (Georgia Tech)
- MuJava Mutation system for Java

http://www.ise.gmu.edu/~offutt/mujava/

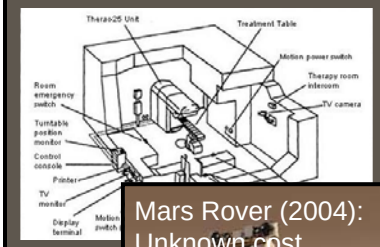
- Mutation Testing Online Resources

http://www.mutationtest.net/twiki/bin/view/Resources/WebHome

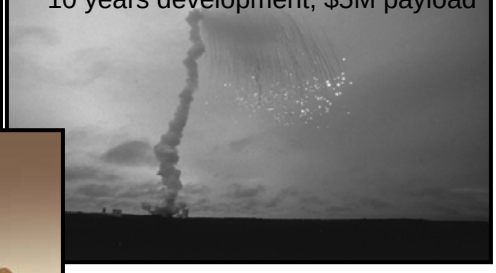
Regression Testing: Selection, Prioritization, Reduction, and Augmentation

High Cost of Software Failure

Therac-25 (1985-87): Deaths



Ariane 5 Explosion (1996): $7B cost, 10 years development, $5M payload

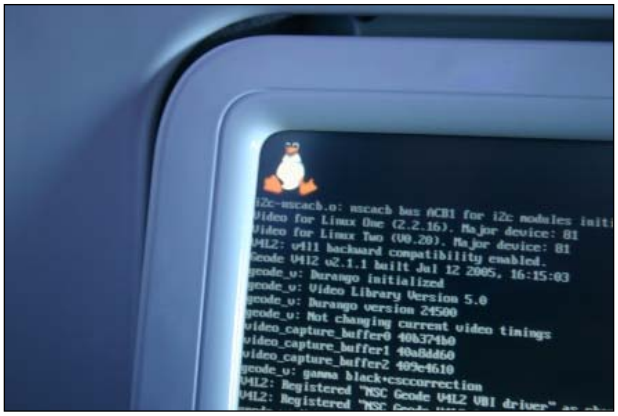


Mars Rover (2004): Unknown cost

High Cost of Software Failure

Airplane entertainment system (2008)

- Failed for me and most passengers
- 16 hour flight—Atlanta to Mumbai



Collaboration With Industry

Common Problem

- Changes require rapid modification and testing for quick release (time to market pressures)
- Causing released software to have many defects

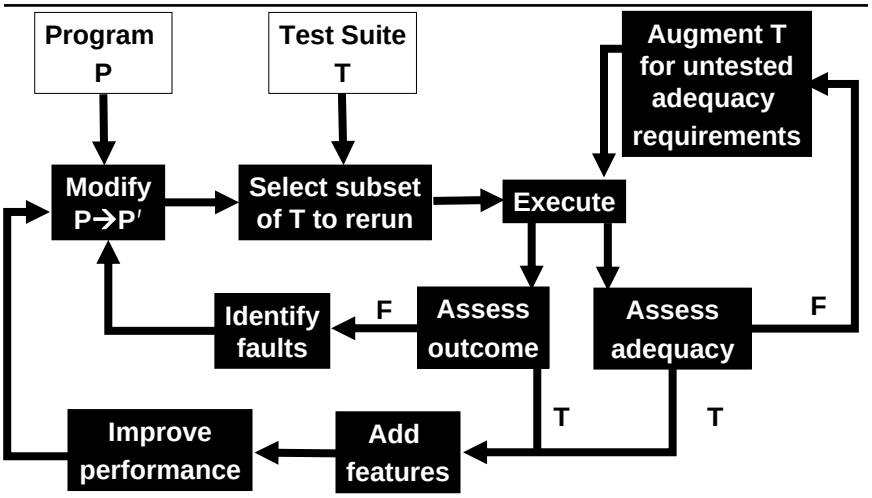
Research Question

How can we test well to gain confidence in the changes in an efficient way before release of changed software?

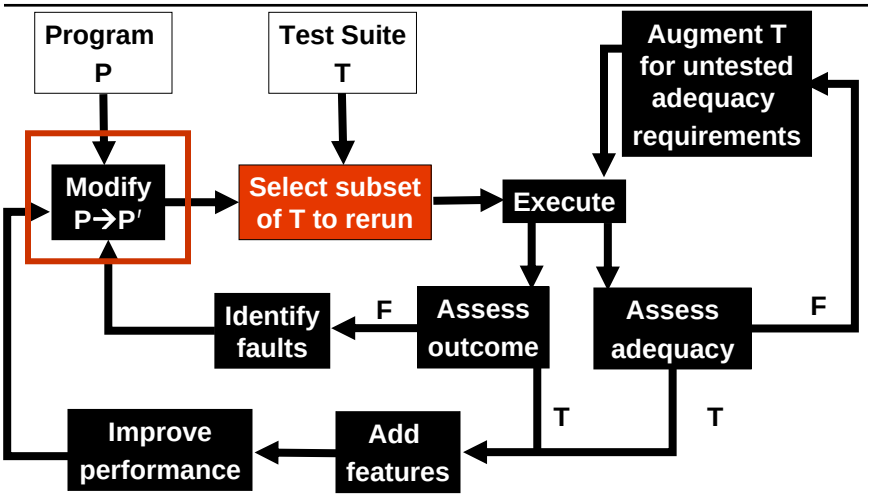
Approach

- Concentrate testing around the changes
- Automate (if possible) the regression testing process

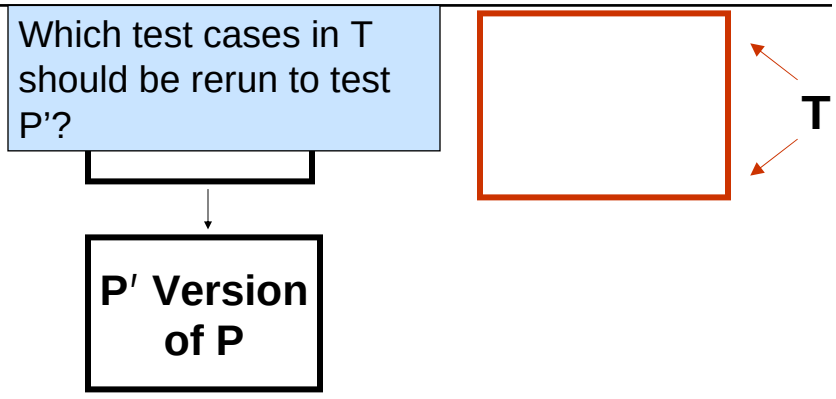
Testing Evolving Software



Select Subset of T to Rerun



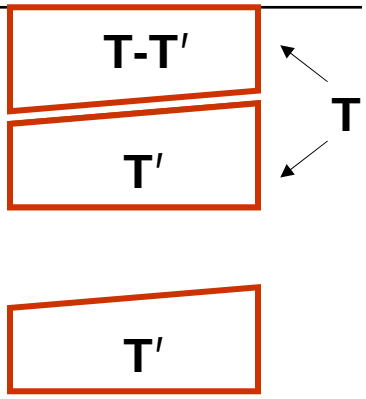
Select Subset of T to Rerun



Select Subset of T to Rerun

Which test cases in T should be rerun to test P'?

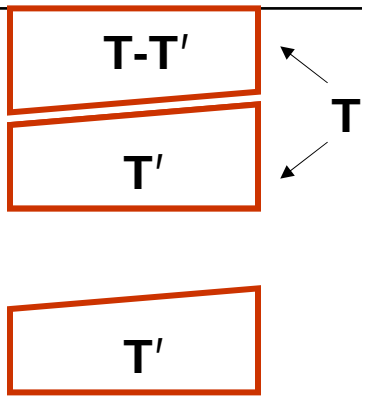
Solution
Partition T into two subsets
- run one on P'
- don't run the other

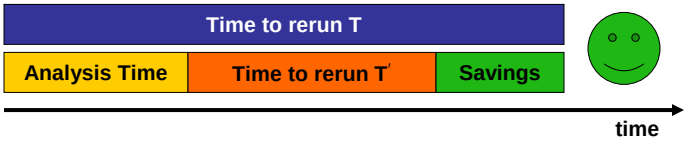


Select Subset of T to Rerun

Which test cases in T should be rerun to test P'?

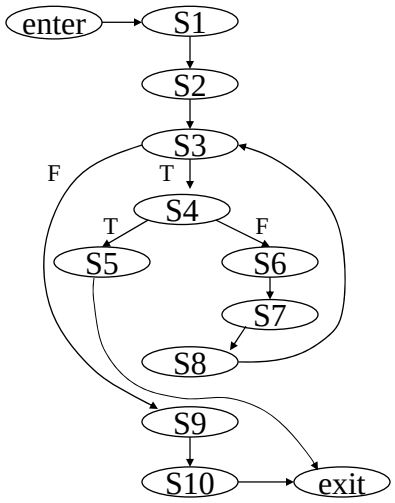
Solution
Partition T into two subsets
- run one on P'
- don't run the other





Regression Test Selection: Create Graph Representation

```
Procedure Avg
S1   count = 0
S2   fread(fptr,n)
S3   while (not EOF) do
S4     if (n<0)
S5       return(error)
       else
S6       nums[count] = n
S7       count++
       endif
S8     fread(fptr,n)
     endwhile
S9   avg = mean(nums,count)
S10 return(avg)
```



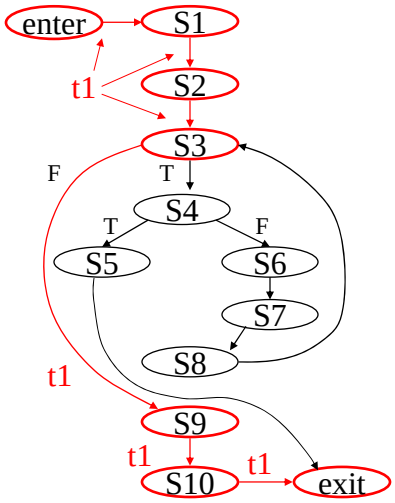
Regression Test Selection: Gather Execution Information

```
Procedure Avg
S1   count = 0
S2   fread(fptr,n)
S3   while (not EOF) do
S4     if (n<0)
S5       return(error)
       else
S6       nums[count] = n
S7       count++
       endif
S8     fread(fptr,n)
     endwhile
S9   avg = mean(nums,count)
S10 return(avg)
```

test	input	output
t1	empty file	0

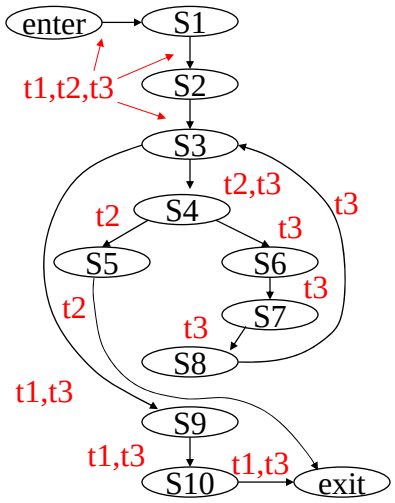
Regression Test Selection: Gather Test History Information

```
Procedure Avg
S1   count = 0
S2   fread(fptr,n)
S3   while (not EOF) do
S4     if (n<0)
S5       return(error)
       else
S6       nums[count] = n
S7       count++
       endif
S8     fread(fptr,n)
     endwhile
S9   avg = mean(nums,count)
S10 return(avg)
```



Regression Test Selection: Gather Test History Information

test	input	output
t1	empty file	0
t2	-1	error
t3	1 2 3	2



Regression Test Selection: Consider P and P'

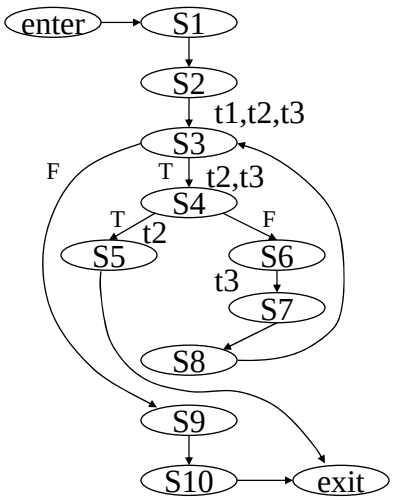
Procedure Avg

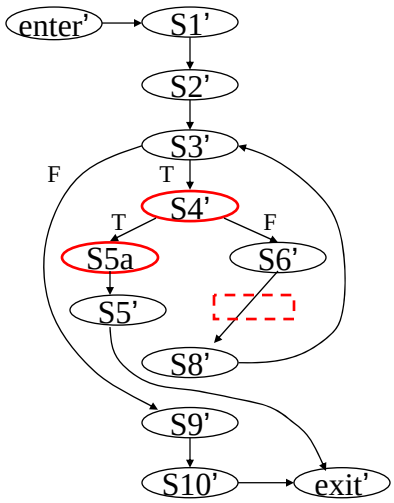
```
S1   count = 0
S2   fread(fptr,n)
S3   while (not EOF) do
S4     if (n<0)

S5       return(error)
       else
S6       nums[count] = n
S7       count++
       endif
S8     fread(fptr,n)
     endwhile
S9   avg = mean(nums,count)
S10 return(avg)
```

Procedure Avg'

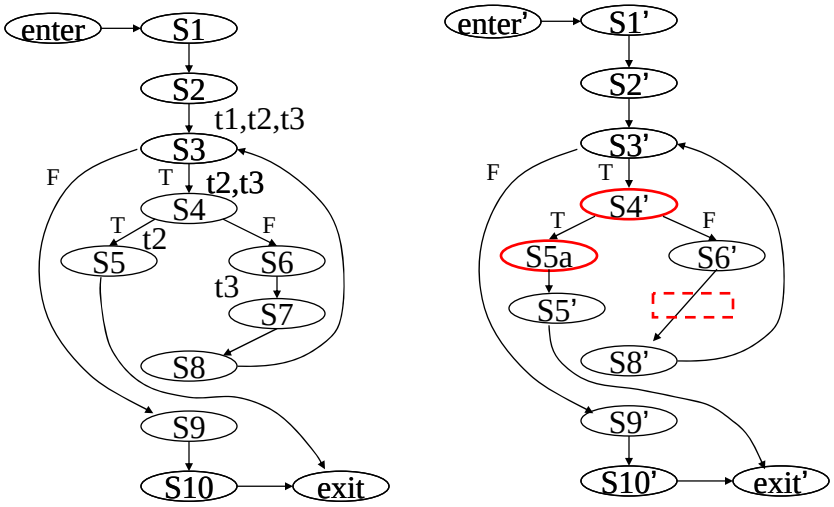
```
S1'   count = 0
S2'   fread(fptr,n)
S3'   while (not EOF) do
S4'     if (n<=0)
S5a       print("input error")
S5'       return(error)
        else
S6'       nums[count] = n
        endif
S8'     fread(fptr,n)
      endwhile
S9'   avg = mean(nums,count)
S10'  return(avg)
```

Regression Test Selection: Consider CFGs for P and P'

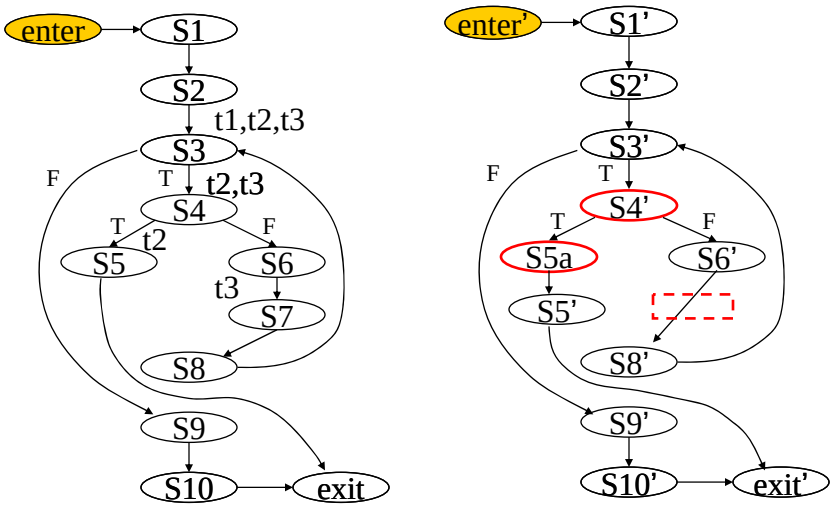




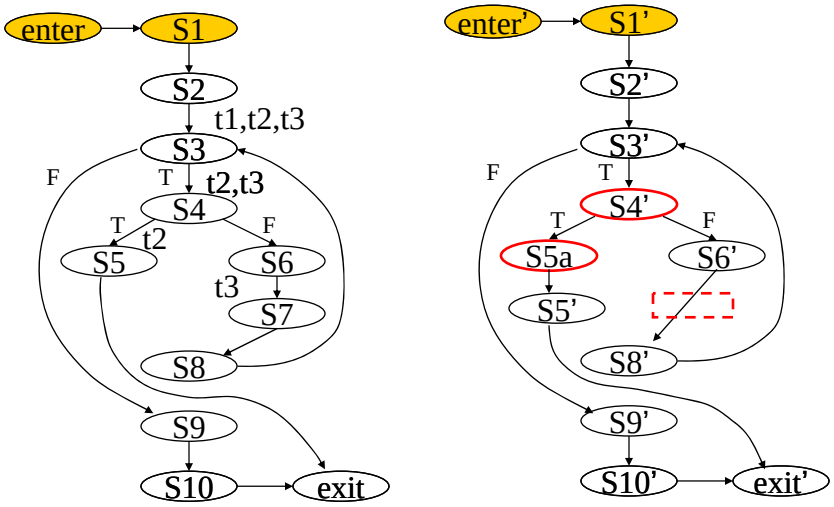
Regression Test Selection: Consider CFGs for P and P'



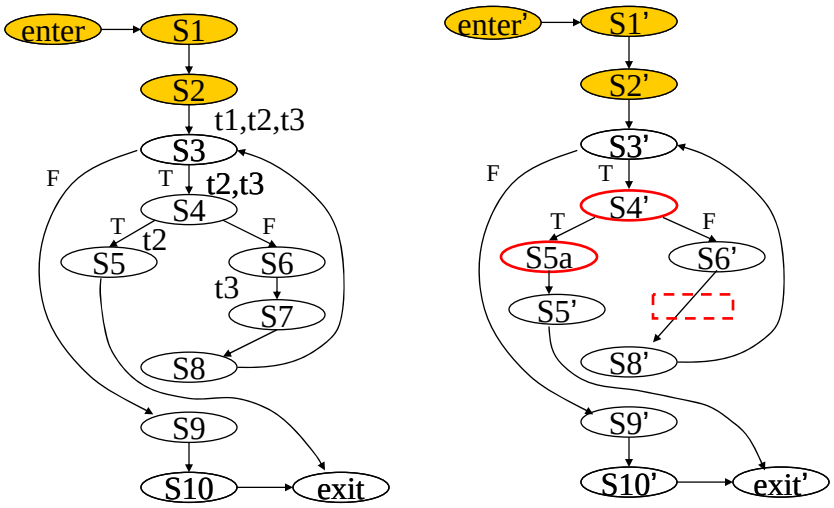
Regression Test Selection: Traverse CFGs for P and P'



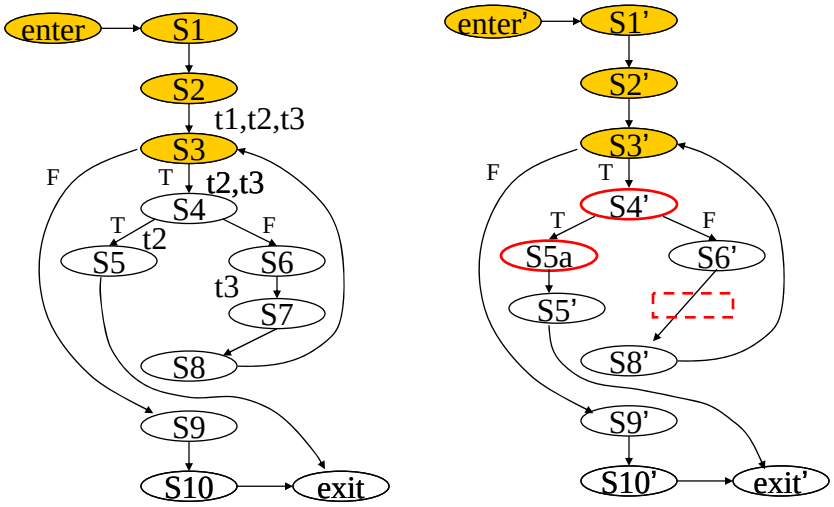
Regression Test Selection: Traverse CFGs for P and P’



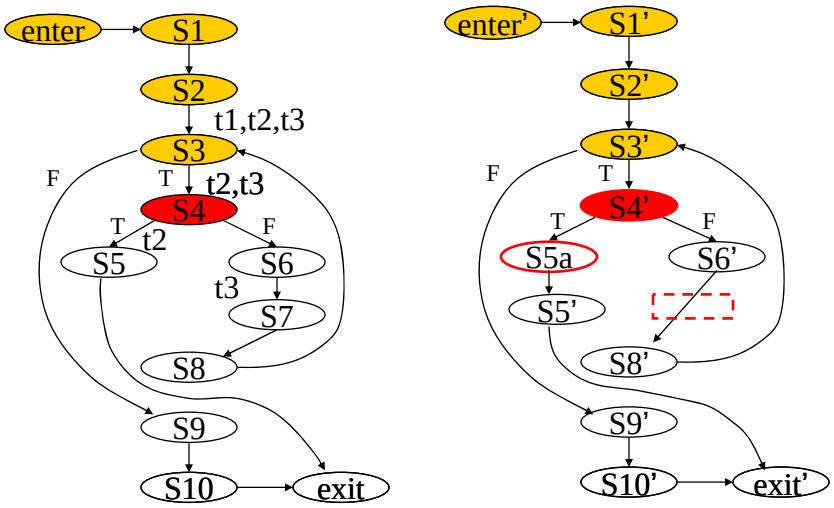
Regression Test Selection: Traverse CFGs for P and P’



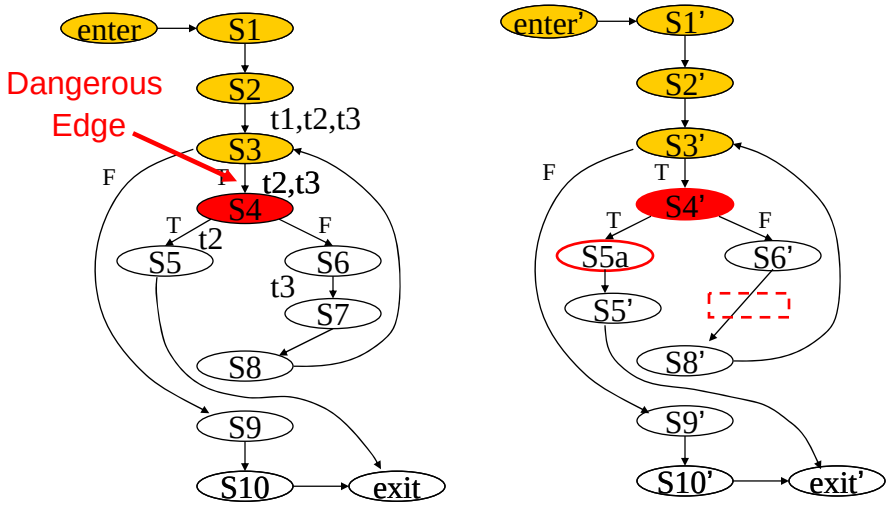
Regression Test Selection: Traverse CFGs for P and P'



Regression Test Selection: Traverse CFGs for P and P'

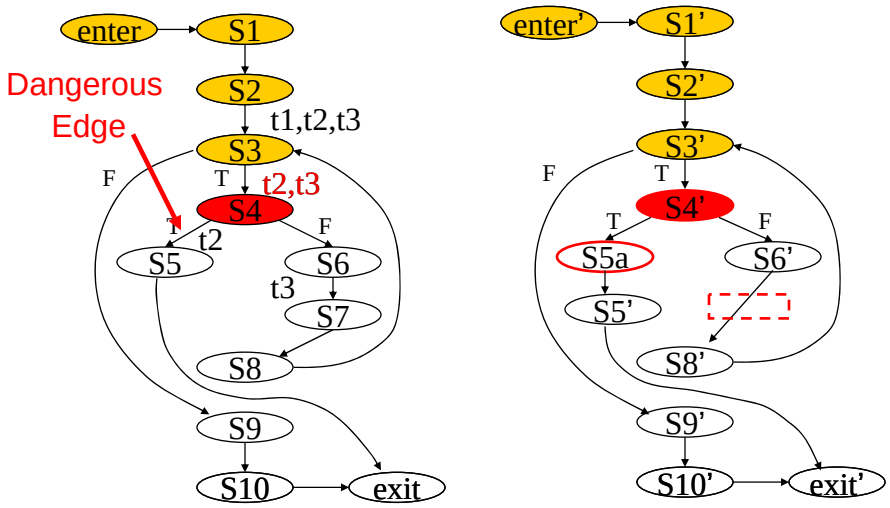


Regression Test Selection: Traverse CFGs for P and P'



Regression Test Selection: Traverse CFGs for P and P'

T' = {t2, t3}



Algorithm DejaVu

```
Input: P, P', T      Output: T'
 1. Build CFGs G and G' for P and P'
 2. Compare(G.EntryNode,G'.EntryNode)

 3. Compare(N,N')
 4.     mark N "N'-visited"; initialize DangerousEdges to empty
 5.     for each pair of successors C and C' of N and N'
 6.         on equivalently labeled edges do
 7.     if C is not marked "C'-visited"
 8.         if C and C' are not lexically identical
 9.             Add (C,C') to DangerousEdges
10.         else
11.             Compare(C,C')
```

Algorithm Efficiency

CFG construction: linear in program size

Graph walk (graph sizes n, n'; test set size t):

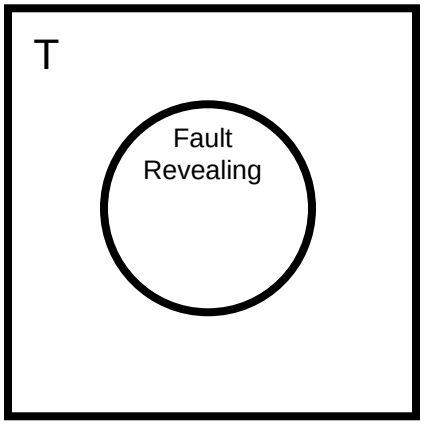
$O (t * n * n')$

(with *multiply-visited nodes)*

$O (t * \min(n,n'))$

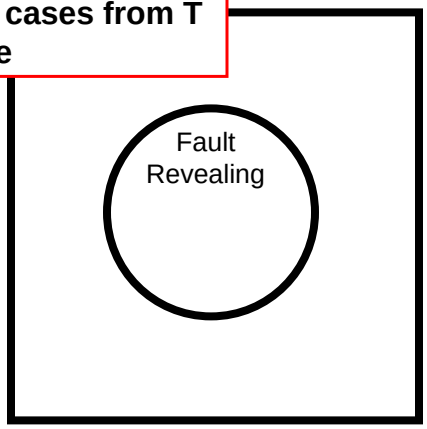
(with no *multiply-visited nodes*)

Precision and Safety

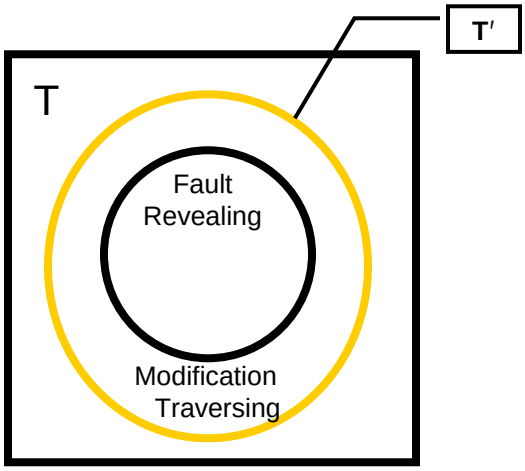


Precision and Safety

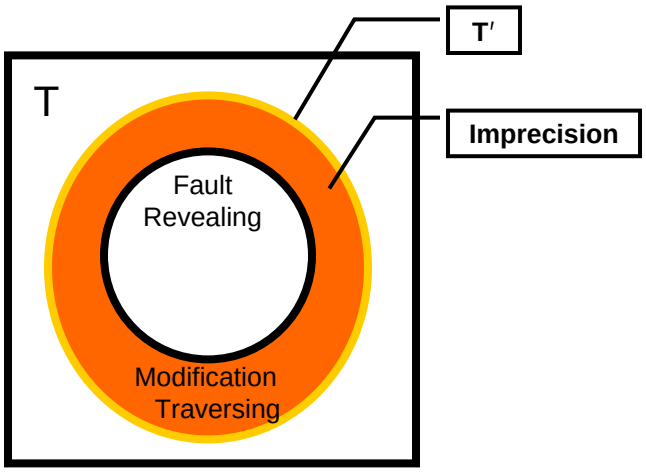
Selecting only fault-revealing test cases from T is undecidable



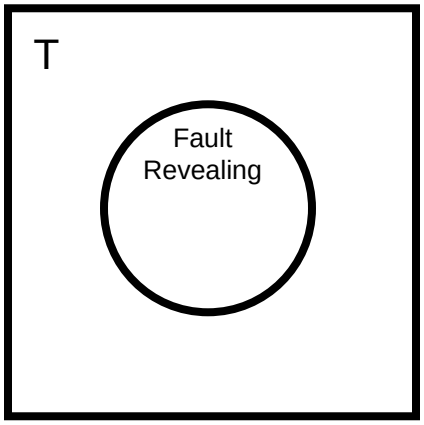
Precision and Safety



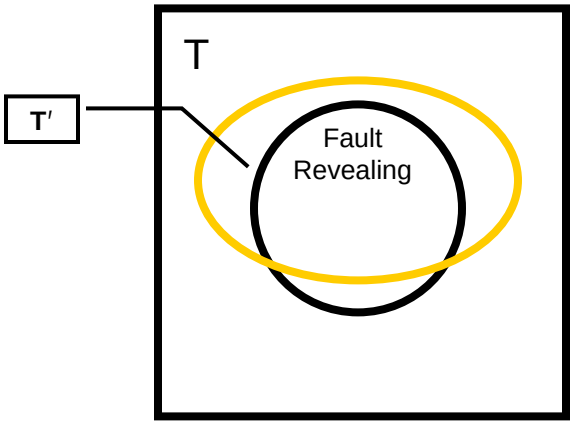
Precision and Safety



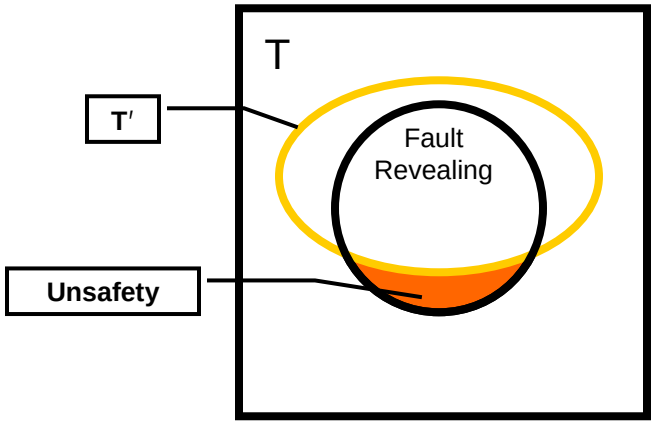
Precision and Safety



Precision and Safety



Precision and Safety



DejaVu Algorithm

- Algorithm needs
  - Graph representation for original P and changed P'
  - Way to associate test cases in T with entities in P
  - Way to differentiate P and P'
- Algorithm is language independent
  - For C, used CFGs and ICFGs
  - For Ada, used CFGs and ICFGs (for Boeing)
  - For Java, used JIG (Java Interclass graph) and graphs that represent library interactions, exceptions, polymorphism, etc (got new name DejaVOO)

DejaVu Algorithm

- Algorithm can be used at various levels
  - Branches in CFG
  - Methods, procedures in program
  - Classes
  - UML diagrams
  - Other representations of program

Evidence of Effectiveness

Empirical studies

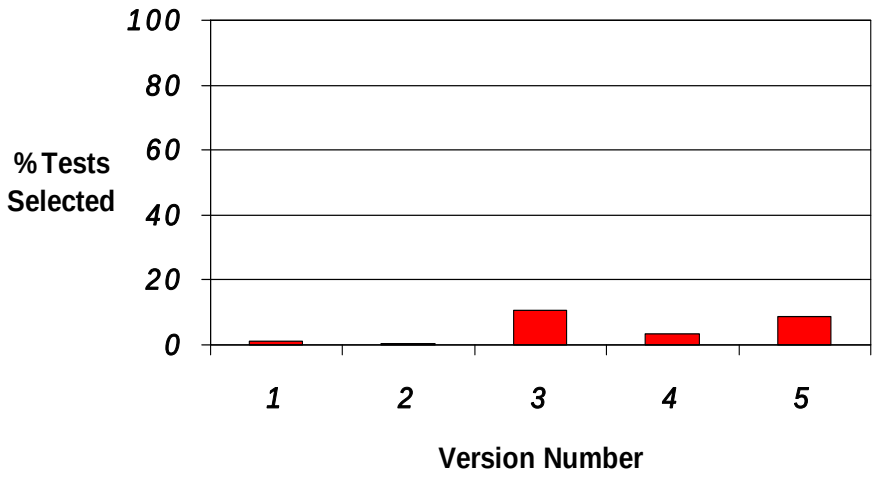
- Empire (C program)
- Coarse vs fine grained (C programs)
- Three Java programs

Study 1: Empire

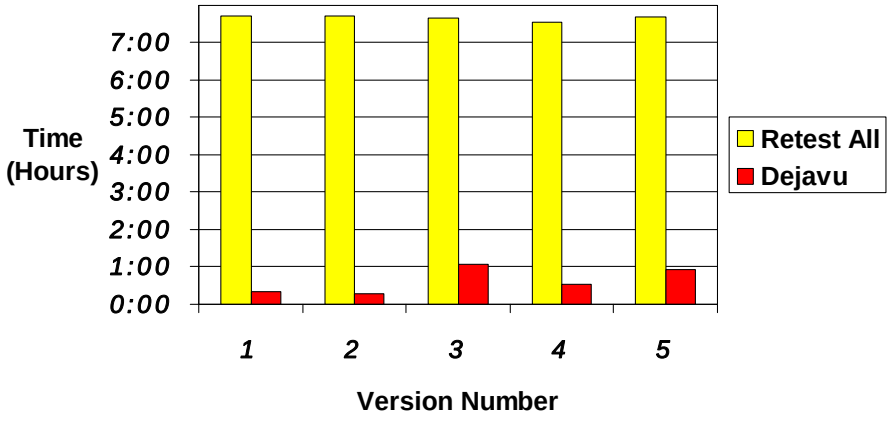
Program	Procs	LOC	Vers	Tests
server	766	49316	5	1033

Version	Functions Modified	LOC Modified
1	3	114
2	2	55
3	11	726
4	11	62
5	42	221

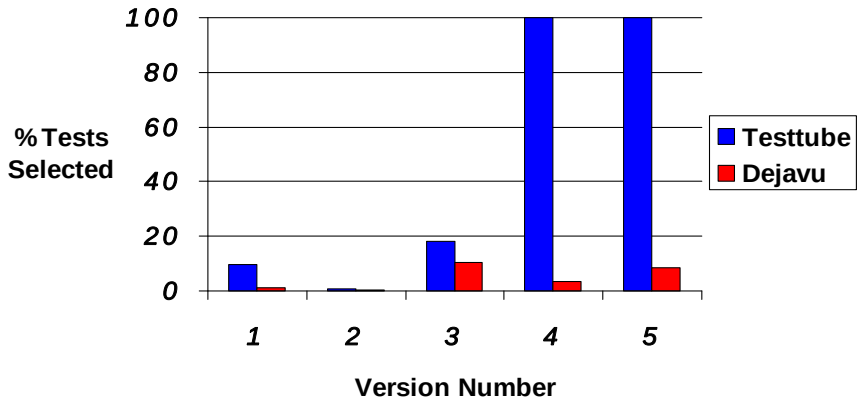
Study 1: Test Selection Percentages



Study 1: Cost Effectiveness



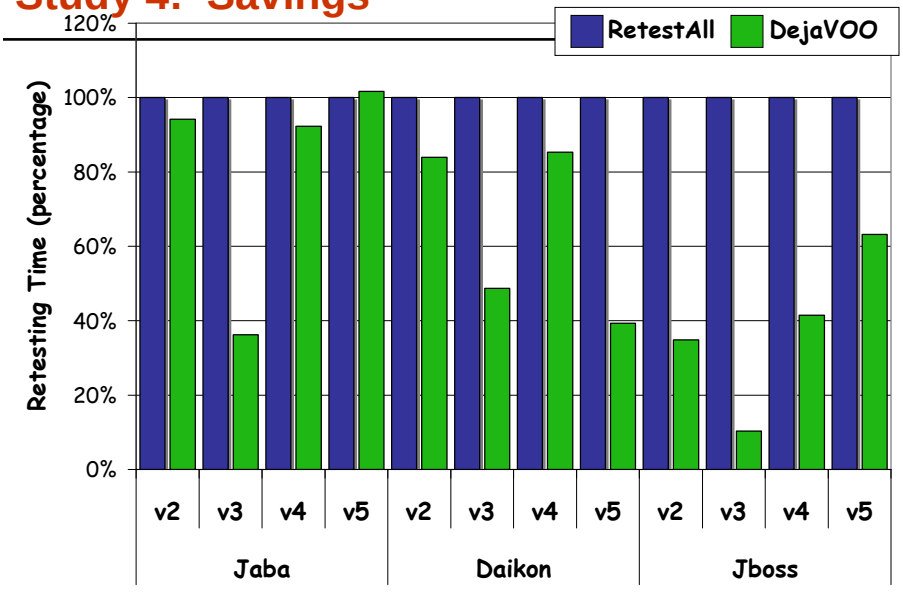
Study 3: Coarse vs Fine Selection

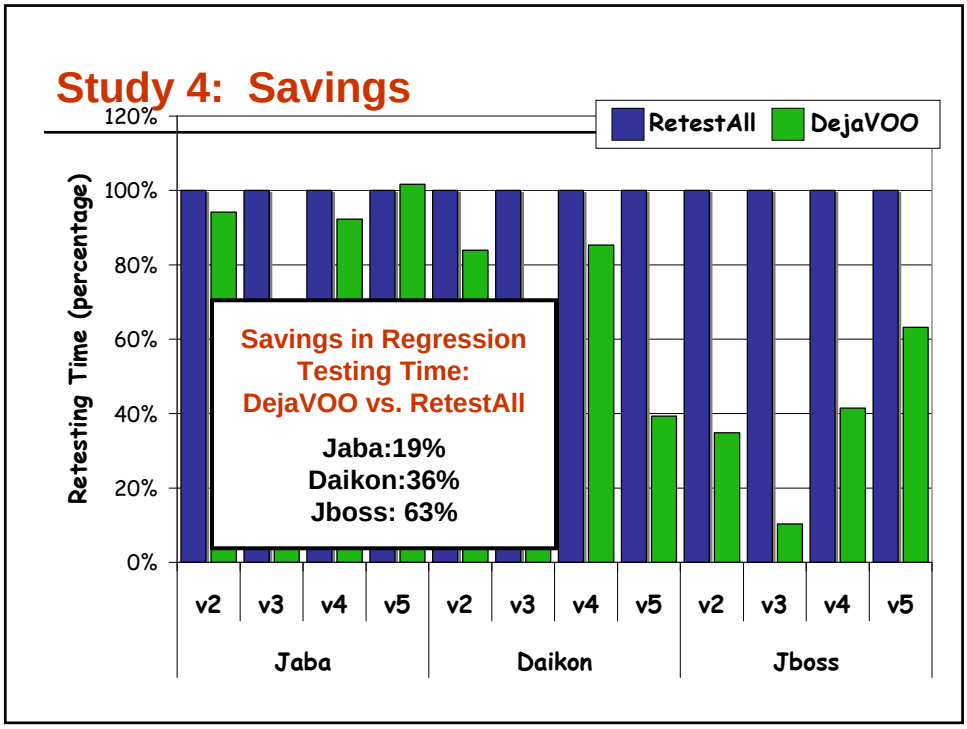


Study 4: Three Large Java Programs

Program	Versions	Classes	KLOC	Test Cases	Retest Time
Jaba	5	525	70	707	54 min
Daikon	5	824	167	200	74 min
Jboss	5	2,403	1,000	639	32 min

Study 4: Savings




Study 4: Savings
RetestAll
DejaVOO
Retesting Time (percentage)
120%
100%
80%
60%
40%
20%
0%
Savings in Regression Testing Time: DejaVOO vs. RetestAll
Jaba:19%
Daikon:36%
Jboss: 63%
v2
v3
v4
v5
Jaba
Daikon
Jboss