Regression Testing

Ajitha Rajan

Evolving Software

Large software systems are usually built incrementally:

- **Maintenance** - fixing errors and flaws, hardware changes
- **Enhancements -** new functionality, improved efficiency, extension, new regulations

`Program v1` --upgrade--> `Program v2` --upgrade--> `Program v3` --upgrade-->

Regressions

- Ideally, software should *improve* over time.
- But changes can both
  - **Improve** software, adding features and fixing bugs
  - **Break** software, introducing new bugs
- We call such breaking changes ***regressions***

Regression Testing

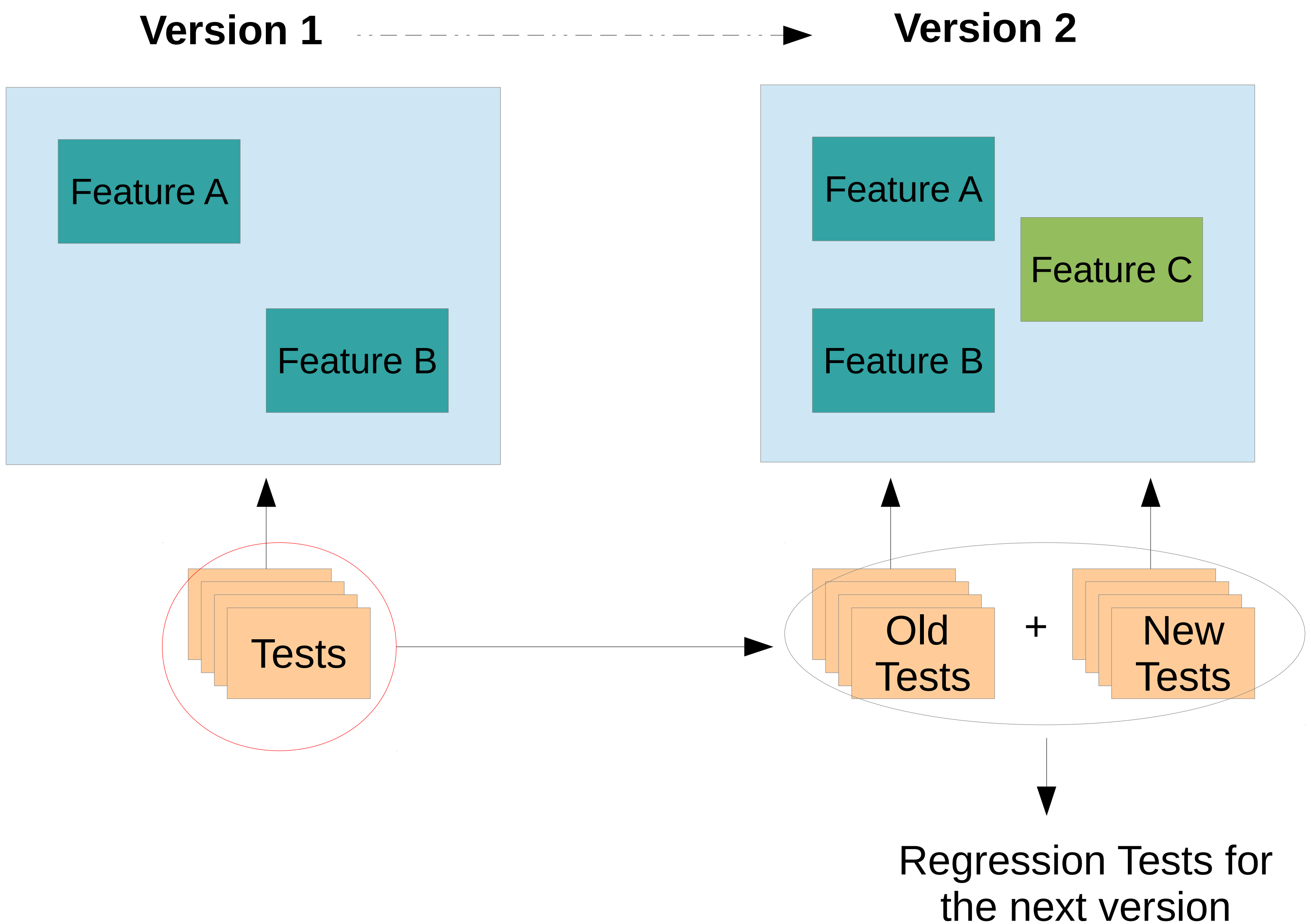
Version 1

1. Develop P
2. Test P
3. Release P

Version 2

4. Modify P to P'
5. Test P' for *new functionality* or *bug fixing*
6. Perform **regression testing** on P'
7.Release P'

Example


Version 1
Version 2
Feature A
Feature B
Feature A
Feature C
Feature B
Tests
Old Tests
+
New Tests
Regression Tests for the next version

Consequences of Poor Regression Testing

- Thousands of 1-800 numbers disabled by a poorly tested software upgrade (December 1991)
- Fault in an SS7 software patch causes extensive phone outages (June 1991)
- Fault in a 4ESS upgrade causes massive breakdown in the AT&T network (January 1990)

AT&T Network Outage, Jan 1990

```
While (ring receive buffer | empty and side buffer | empty)
{
Initialize pointer to first message in side buffer or ring received buffer
Get a copy of buffer
Switch (message) {
Case incoming message: if (sending switch = out of service)
        {
        if (ring write buffer = empty)
        Send in service to states map manager;
        Else
        Break;
        }
Process incoming message, set up pointers to optional parameters
        Break;
        :
        }
Do optional parameter work
}
```

Regression

- Yesterday it worked, today it doesn't.
  - I was fixing X, and accidentally broke Y
- Tests must be re-run after any change
  - Adding new features
  - Changing, adapting software to new conditions
  - Fixing other bugs
- Regression testing can be a major cost of software maintenance
  - Sometimes much more than making the change

Regression Testing takes too long

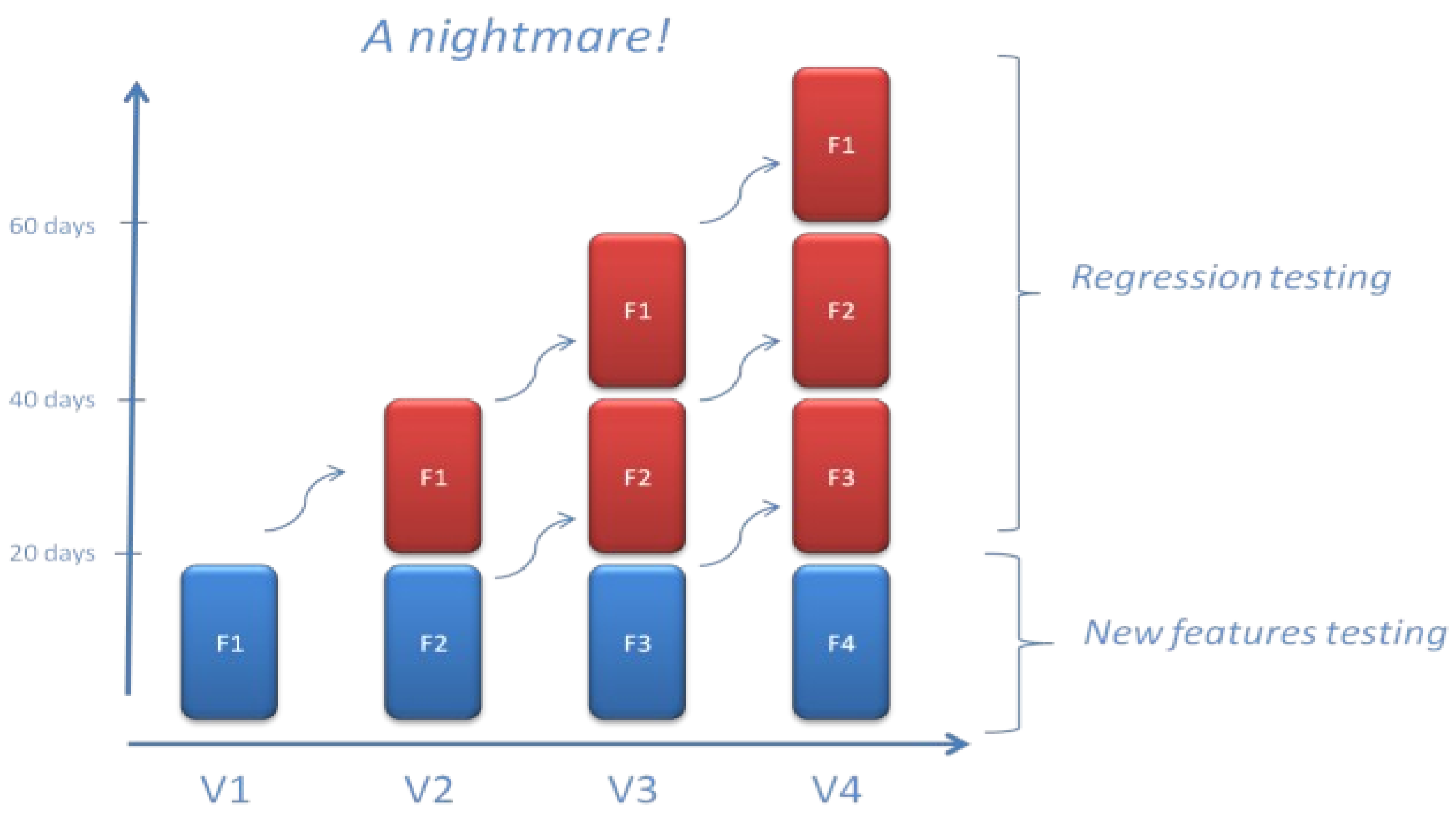


Image from http://blog.kalistick.com/tools/improving-regression-testing-effectiveness/

Basic Problems of Regression Test

- Maintaining test suite
  - If I change feature X, how many test cases must be revised because they use feature X?
  - Which test cases should be removed or replaced? Which test cases should be added?
- Cost of re-testing
  - Often proportional to product size, not change size
  - Big problem if testing requires manual effort
  - Possible problem even for automated testing, when the test suite and test execution time grows beyond a few hours

Test Case Maintenance

Some maintenance is inevitable
If feature X has changed, test cases for feature X will require updating

Some maintenance should be avoided
Example: Trivial changes to user interface or file format should not invalidate large numbers of test cases

Test suites should be modular!
Avoid unnecessary dependence

Generating concrete test cases from test case specifications can help

Obsolete and Redundant

- Obsolete: A test case that is no longer valid
  - Should be removed from the test suite
- Redundant: A test case that does not differ significantly from others
  - Unlikely to find a fault missed by similar test cases
  - Has some cost in re-execution
  - May or may not be removed, depending on costs

Regression Test Optimization

- Re-test All
- Regression Test Selection
- Regression Test Set Minimisation
- Regression Test Set Prioritisation

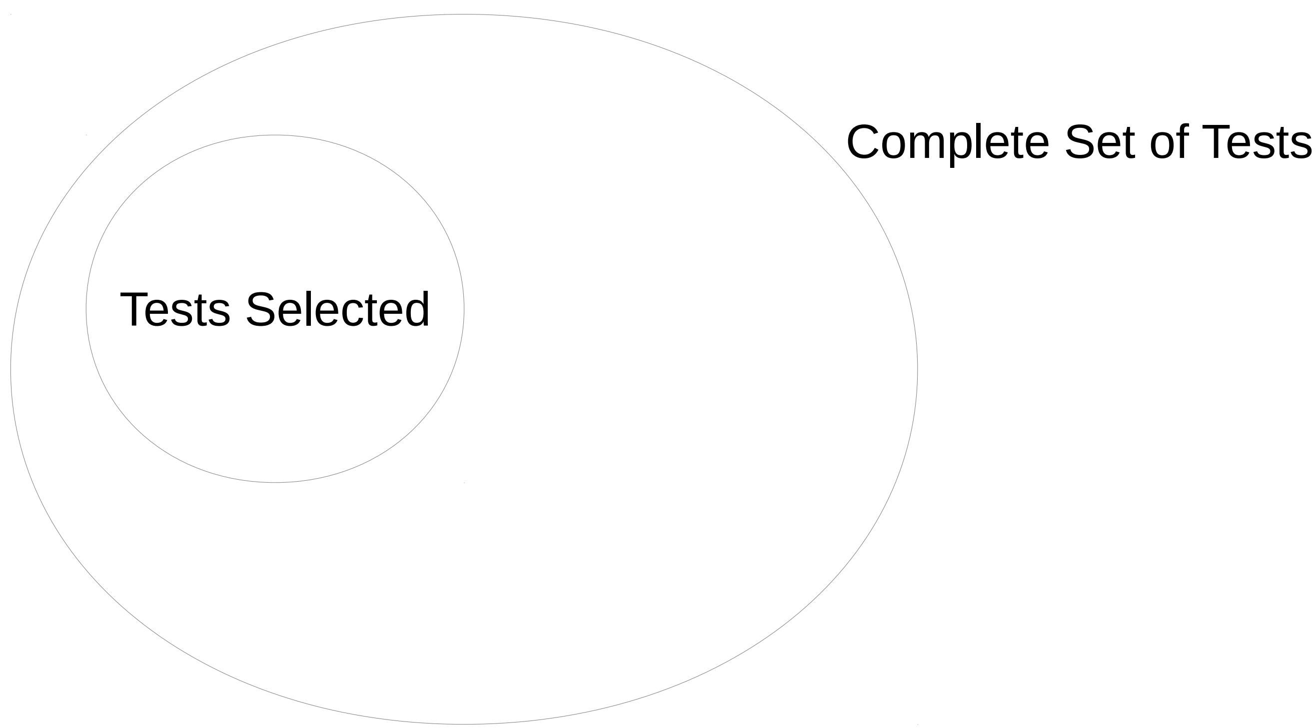
Re-test All Approach

Too Expensive!

- Traditional Approach – Select All
- The test-all approach is good when you want to be certain that the new version works on all tests developed for the previous version.
- What if you only have limited resources to run tests and have to meet a deadline?
- Those on which the new and the old programs produce different outputs (Undecidable)

Regression Test Selection

From the entire test suite, only select subset of test cases whose execution is relevant to changes



Code-based Regression Test Selection

- Observation: A test case can’t find a fault in code it doesn’t execute
  - In a large system, many parts of the code are untouched by many test cases
- So: Only execute test cases that execute changed or new code

Control-flow and Data-flow Regression Test Selection

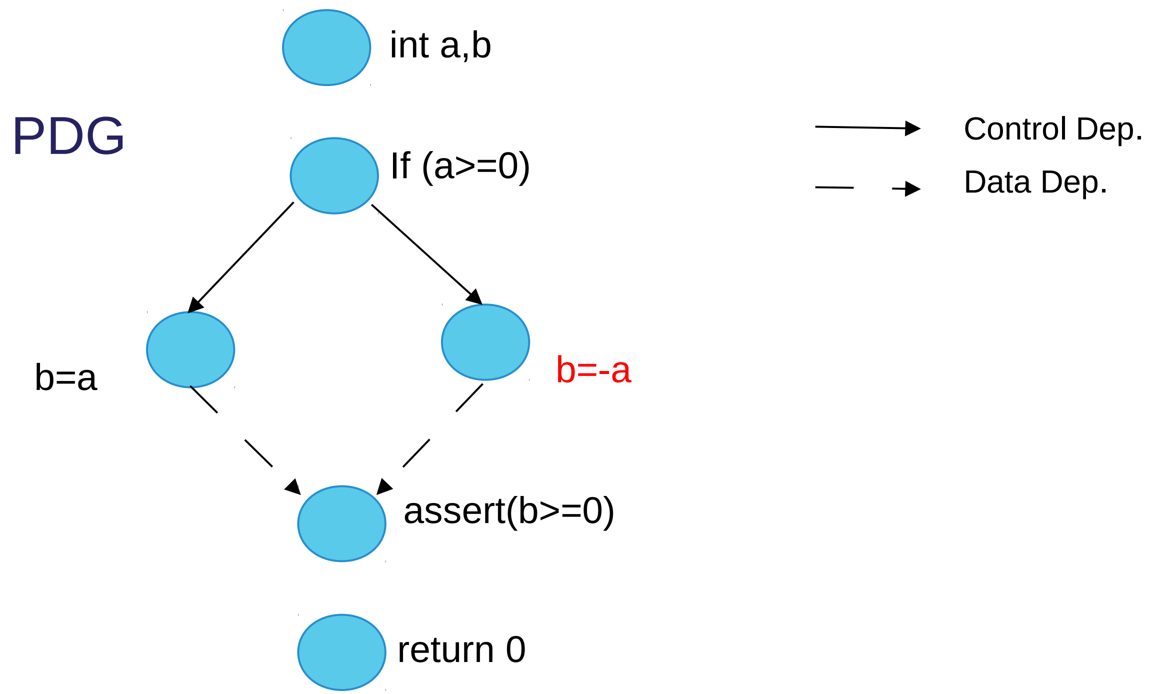
- Same basic idea as code-based selection
  - Re-run test cases only if they include changed elements
  - Elements may be modified control flow nodes and edges, or definition-use (DU) pairs in data flow
- To automate selection:
  - Tools record elements touched by each test case
    - Stored in database of regression test cases
  - Tools note changes in program
  - Check test-case database for overlap

Specification-based Regression Test Selection

- Like code-based and structural regression test case selection
  - Pick test cases that test new and changed functionality
- Difference: No guarantee of independence
  - A test case that isn't "for" changed or added feature X might find a bug in feature X anyway
- Typical approach: Specification-based prioritization
  - Execute all test cases, but start with those that related to changed and added features

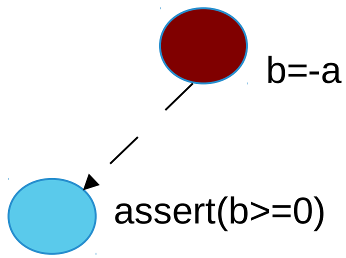
Example

```
int main()
{
        int a, b;
        if (a>=0)
                b = a;
        else
                b = -a;
        assert(b >= 0);
        return 0;
}
```



Forward Slice

Depth first traversal from node b = -a;



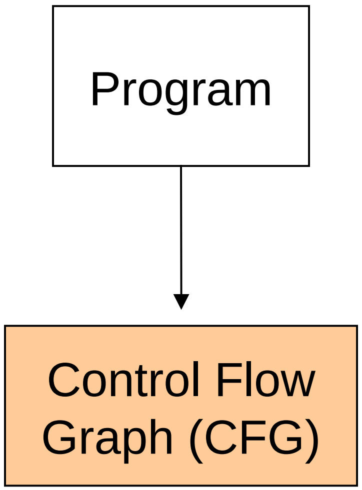
Slicing procedure

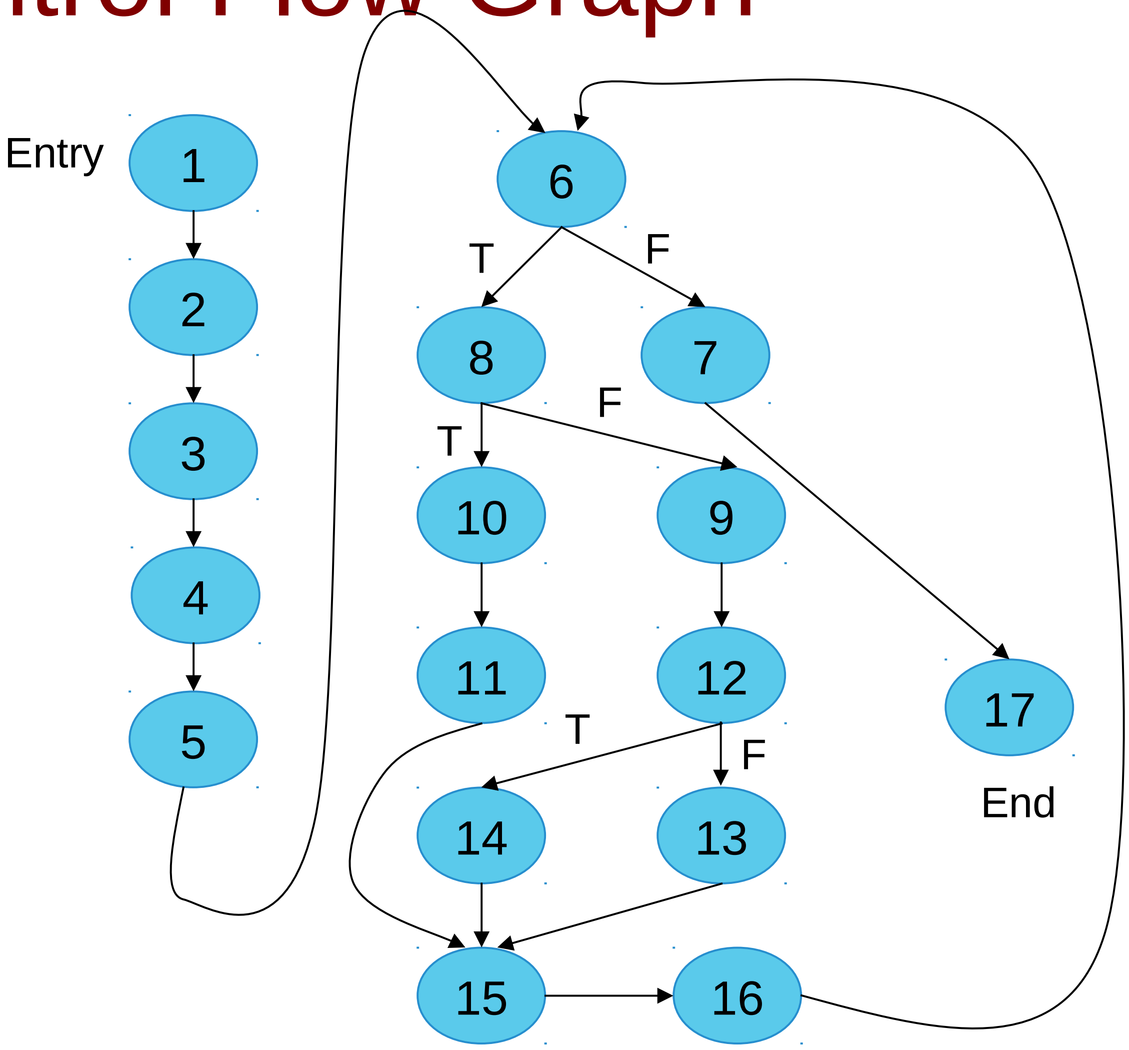
Program

Computing the greatest in an array of integers

```
int main(int argc, char* argv[]) {
  unsigned int num[5] = {12, 23, 4, 78, 34};
  unsigned int largest, counter = 0;
  while (counter < 5) {
            if (counter ==0)
                        largest = num[counter];
            else if(largest > num[counter])
                        largest = num[counter];
            ++counter;
  }
  for (counter = 0; counter < 5; counter++)
            assert(largest >= num[counter];
}
```

Construct Control Flow Graph





Build a PDG

- Build a Program Dependence Graph (PDG) that captures control and data dependencies between nodes in CFG

Sample Data Dependency

For *counter* variable

1 → 2,3,4,5,6,7

7 → 2,3,4,5,6,7

```
int main(int argc, char* argv[]) {
  1  unsigned int num[5] = {12, 23, 4, 78, 34},
largest,   counter = 0;
  2  while (counter <5) {
  3          if (counter ==0)
  4                  largest = num[counter];
  5          else if(largest < num[counter])
  6                  largest = num[counter];
  7          counter = counter +1;
      }
    }
```

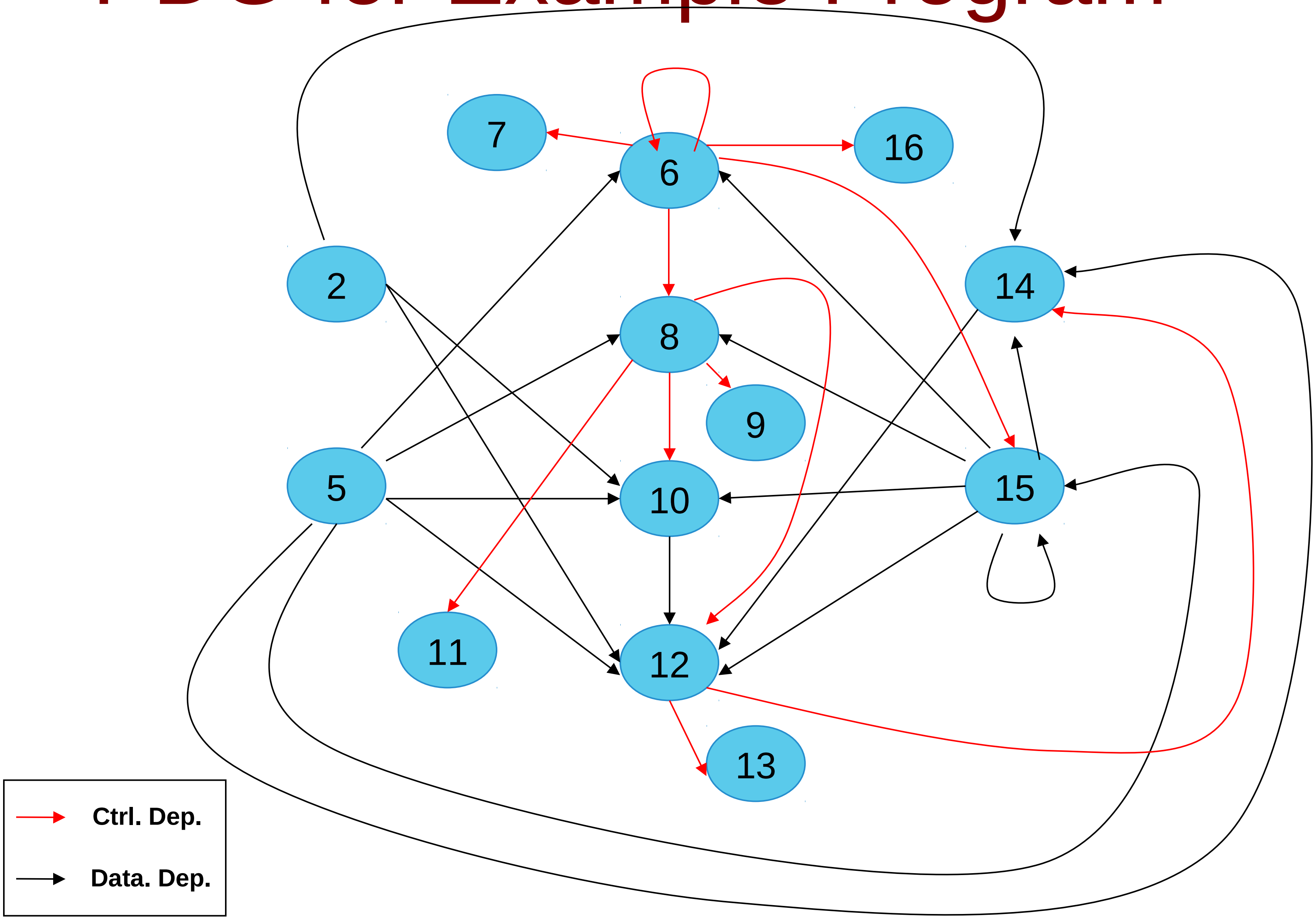
Sample Control Dependency

Conditional in statement 3

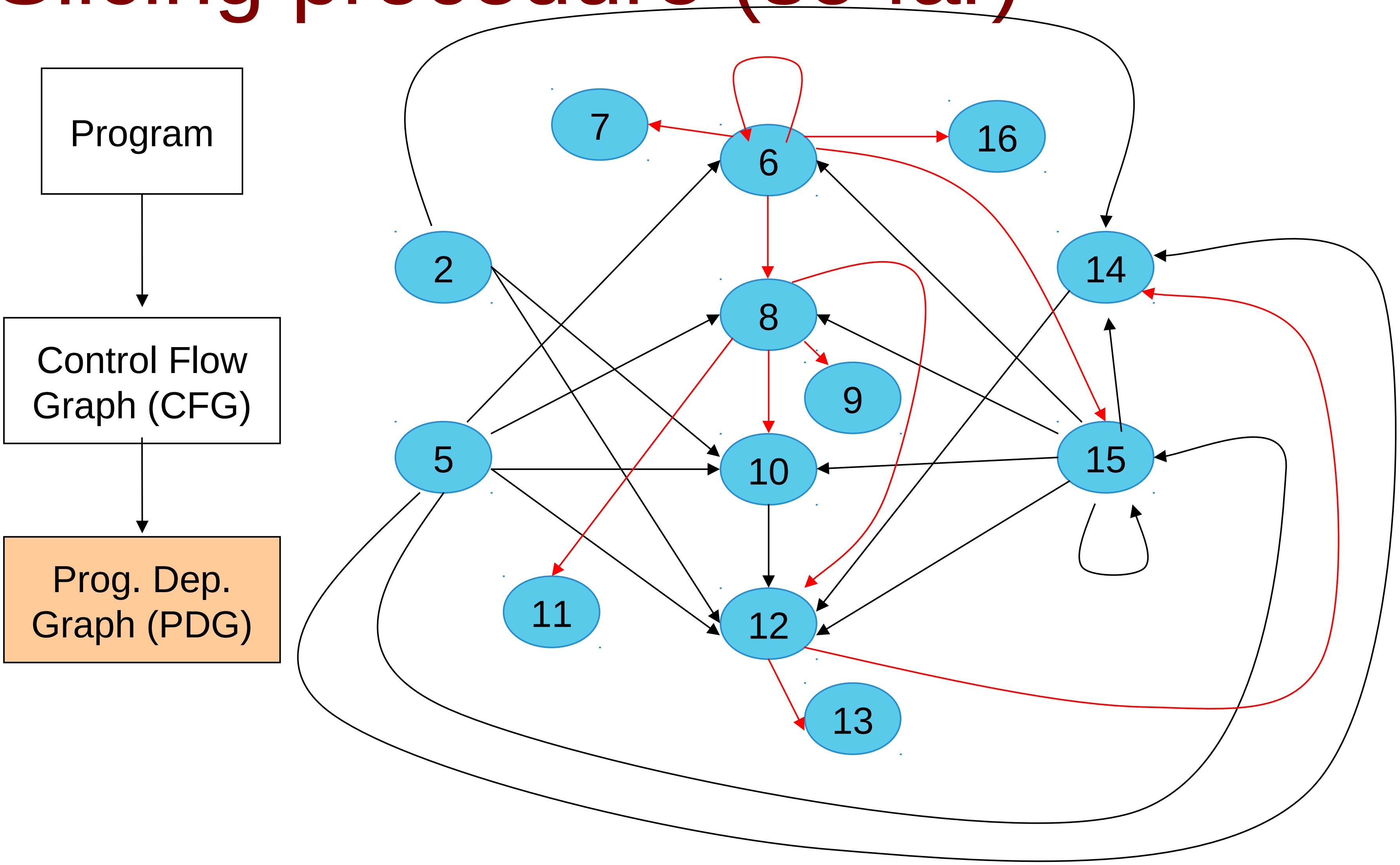
3 → 4, 5

```
int main(int argc, char* argv[]) {
  1  unsigned int num[5] = {12, 23, 4, 78, 34},
largest,   counter = 0;
  2  while (counter <5) {
  3          if (counter ==0)
  4                      largest = num[counter];
  5          else if(largest < num[counter])
  6                      largest = num[counter];
  7          counter = counter +1;
       }
     }
```

PDG for Example Program



Slicing procedure (so far)



Slight change in the example

```
int main(int argc, char* argv[]) {
  unsigned int num[5] = {12, 23, 4, 78, 34},
            largest,   counter = 0;
  while (counter <5) {
            if (counter ==0)
                        largest = num[counter];
            else if(largest > num[counter])
                        largest = num[counter];
            ++counter;
  }
  for (counter = 0; counter < 5;
                        counter++)
            assert(largest >= num[counter];
}
```

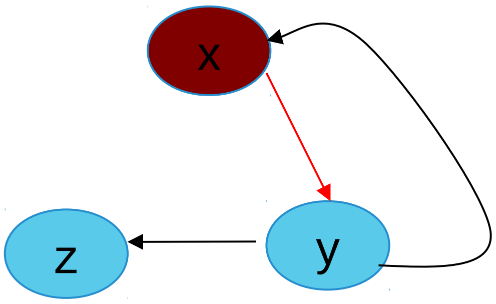
Changed program

Forward Slicing from Changes

- Compute the nodes corresponding to changed statements in the PDG, and
- Compute a transitive closure over all forward dependencies (control + data) from these nodes.

Forward Slice

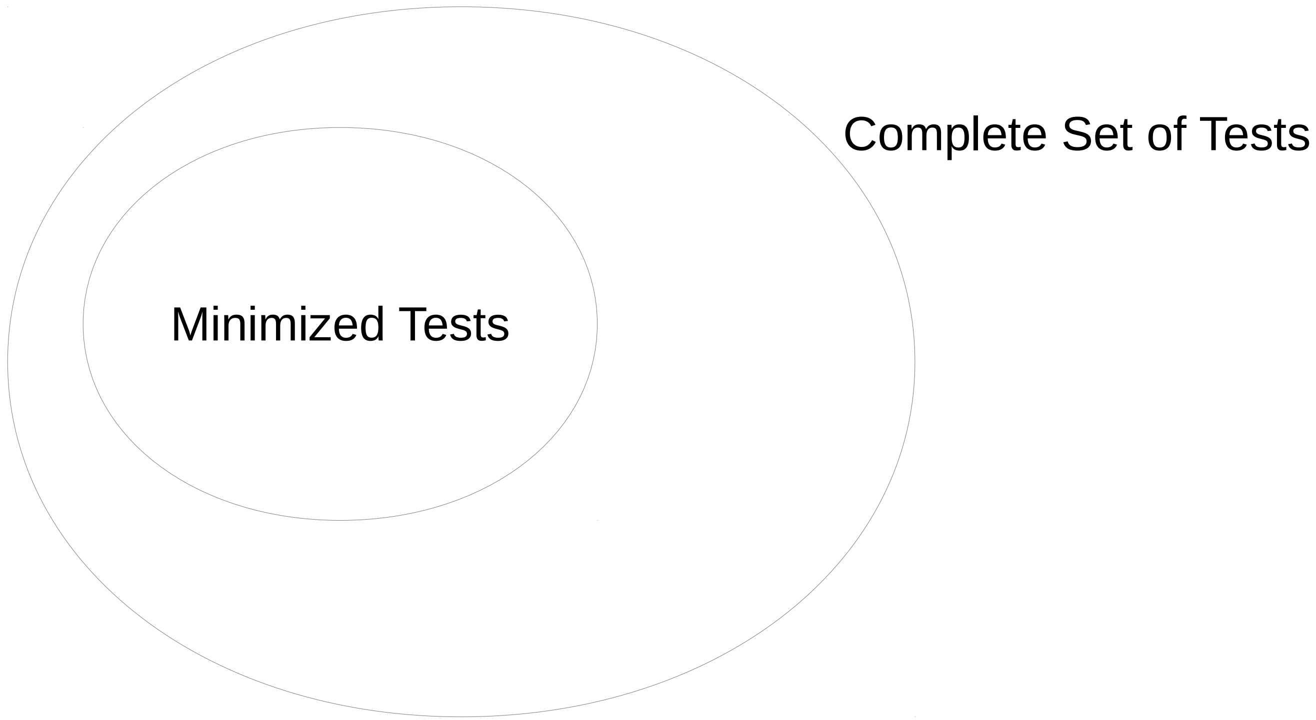
Depth first traversal from changed node



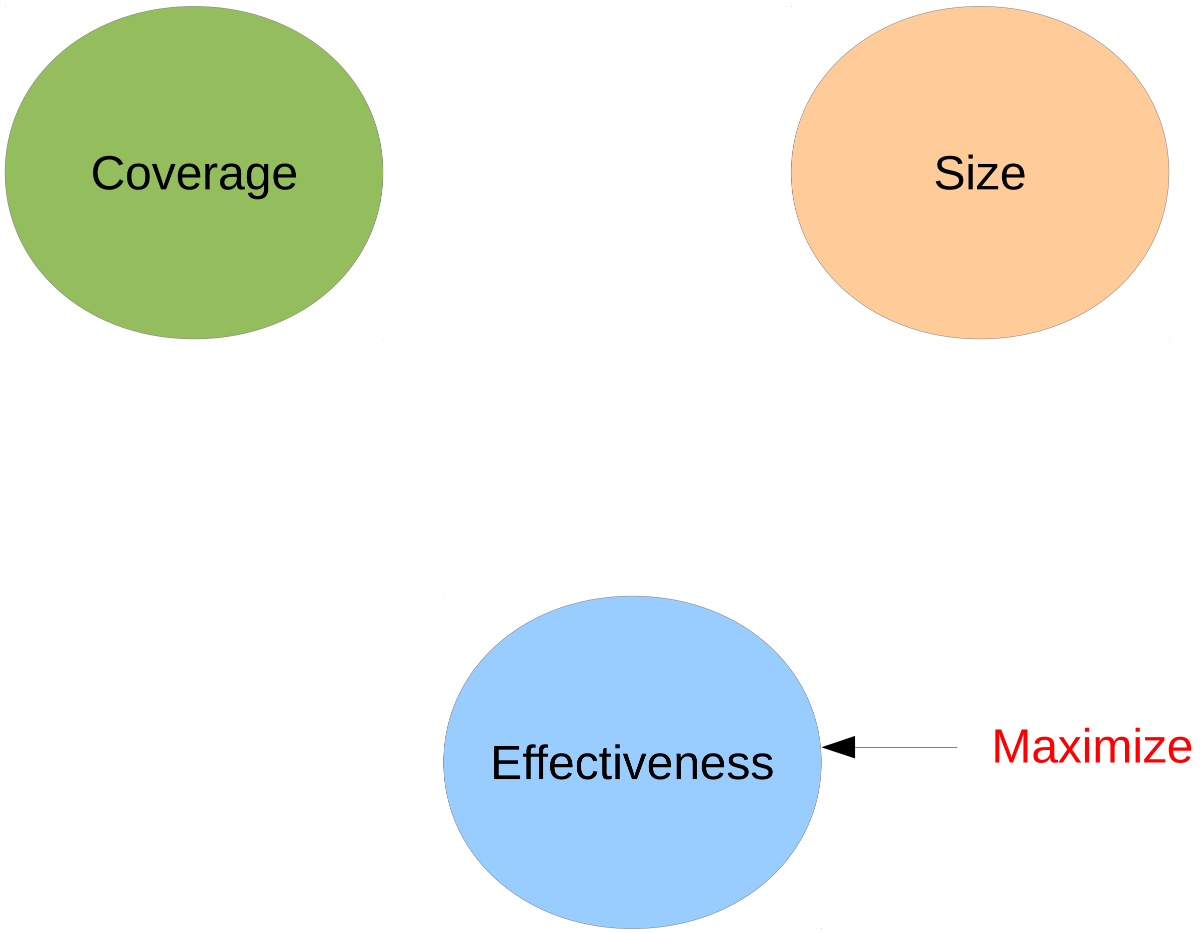

```
else if(largest < num[counter])
        largest = num[counter];
assert (largest >= num[counter];
```

Test Set Minimization

Identify test cases that are redundant and remove them from the test suite to reduce its size.



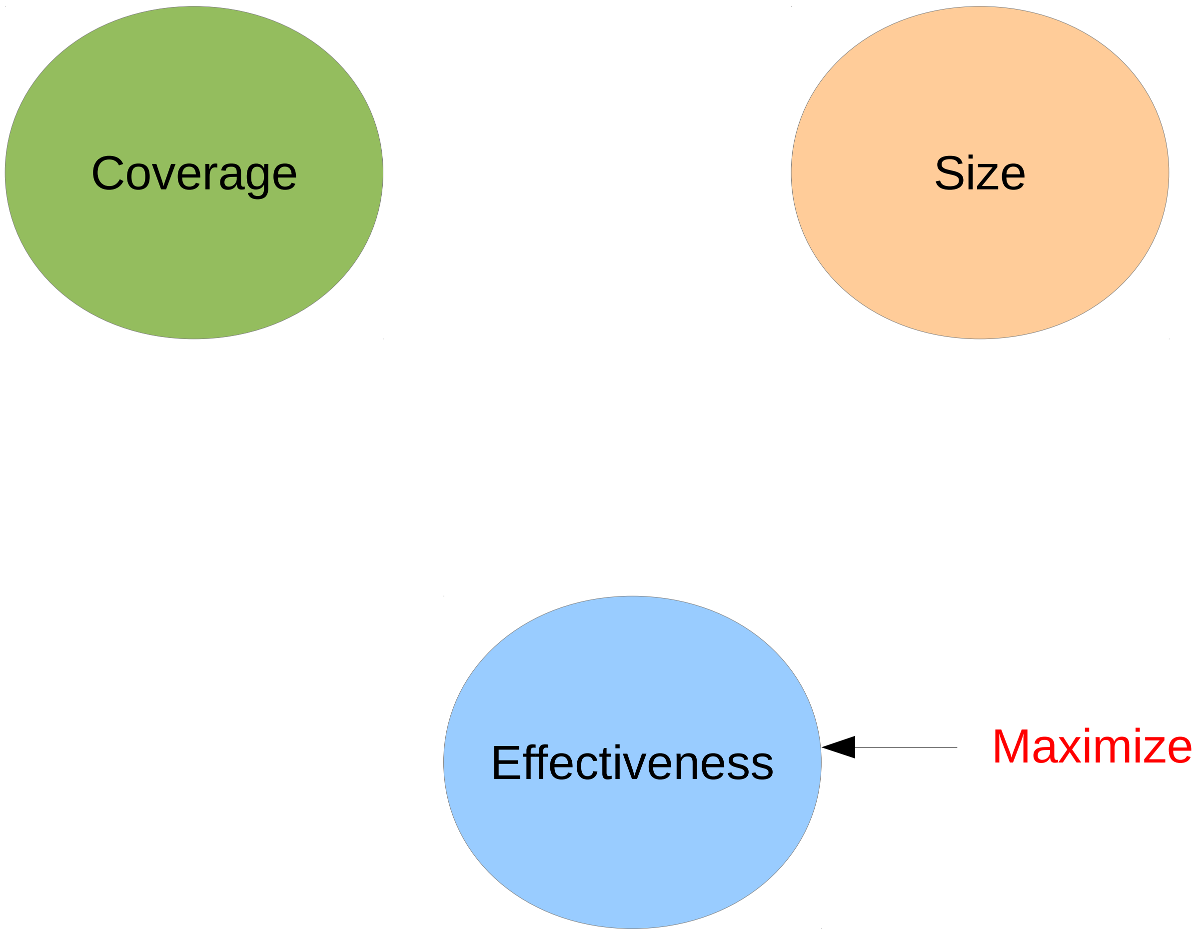
Test Set Attributes



Structural Coverage

- (In)adequacy criteria
  - If significant parts of program structure are not tested, testing is surely inadequate
- Control flow coverage criteria
  - Statement (node, basic block) coverage
  - Branch (edge) coverage
  - Condition coverage
  - Path coverage
  - Data flow (syntactic dependency) coverage
- Attempted compromise between the impossible and the inadequate

Test Set Attributes



Test Set Attributes

- Higher Coverage -----> Better Fault Detection

- Bigger Size ------> Better Fault Detection

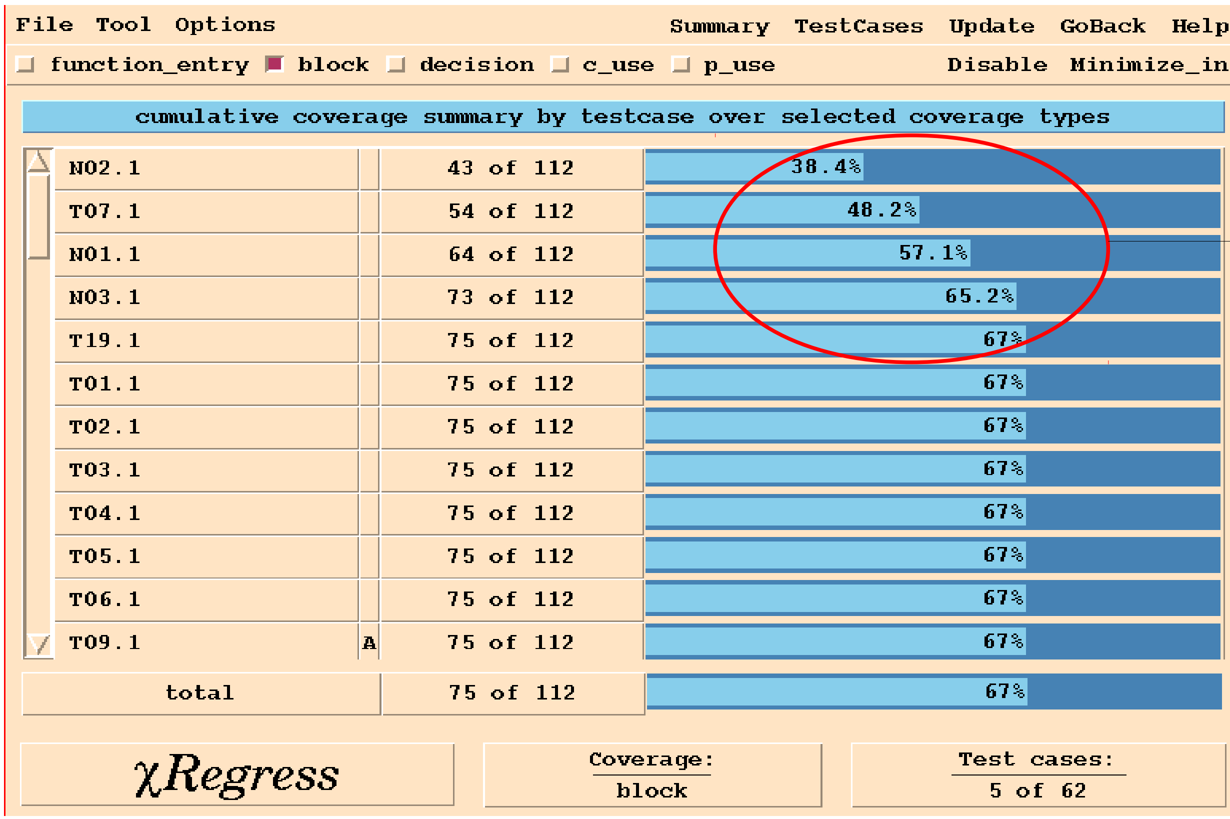
Better Correlated!

Test Set Minimization

- Maximize coverage with minimum number of test cases
- The minimization algorithm can be exponential in time
- Does not occur in our experience
  - Some examples
    - an object-oriented language compiler (100 KLOC)
    - a provisioning application (353 KLOC) with 32K regression tests
    - a database application with 50 files (35 KLOC)
    - a space application (10 KLOC)
- Stop after a pre-defined number of iterations
- Obtain an approximate solution by using a greedy heuristic

Example

Sort test cases in order of increasing cost per additional coverage



Only 5 of the 62 test cases are included in the minimized subset which has the same block coverage as the original test set.

Test Set Prioritisation

- Sort test cases in order of increasing cost per additional coverage
- Select the first test case
- Repeat the above two steps until n test cases are selected or max cost is reached (whichever is first)

Example

- Individual decision coverage and cost per test case

```
$ atac -K -md main.atac wc.atac wordcount.trace

  cost    % decisions    test
  ------ ------------- -------------
  120     57(20/35)      wordcount.1
  50      11(4/35)       wordcount.2
  20      49(17/35)      wordcount.3
  10      11(4/35)       wordcount.4
  40      71(25/35)      wordcount.5
  60      60(21/35)      wordcount.6
  80      11(4/35)       wordcount.7
  20      66(23/35)      wordcount.8
  10      66(23/35)      wordcount.9
  70      60(21/35)      wordcount.10
  50      60(21/35)      wordcount.11
  50      60(21/35)      wordcount.12
  50      20(7/35)       wordcount.13
  40      14(5/35)       wordcount.14
  60      60(21/35)      wordcount.15
  20      26(9/35)       wordcount.16
  150     54(19/35)      wordcount.17
  900     100(35)        == all ==
```

Example

- Prioritized cumulative decision coverage and cost per test case

```
$ atac -Q -md main.atac wc.atac wordcount.trace

cost    % decisions     test
(cum)   (cumulative)
------  ------------    ------------
10      66(23/35)       wordcount.9
30      77(27/35)       wordcount.3
40      83(29/35)       wordcount.4
60      89(31/35)       wordcount.8
100     91(32/35)       wordcount.5
140     94(33/35)       wordcount.14
200     97(34/35)       wordcount.15
280     100(35)         wordcount.7
300     100(35)         wordcount.16
350     100(35)         wordcount.2
400     100(35)         wordcount.12
450     100(35)         wordcount.11
500     100(35)         wordcount.13
560     100(35)         wordcount.6
630     100(35)         wordcount.10
750     100(35)         wordcount.1
900     100(35)         wordcount.17
```

Cost per additional coverage

10/23 = 0.43

(30-10)/(27-23) = 20/4 = 5.00

(40-30)/(29-27) = 10/2 = 5.00

(60-40)/(31-29) = 20/2 = 10.00

(100-60)/(32-31) = 40/1 = 40.00

Increasing Order

Prioritized Rotating Selection

- Basic idea:
  - Execute all test cases, eventually
  - Execute some sooner than others
- Possible priority schemes:
  - Round robin: Priority to least-recently-run test cases
  - Track record: Priority to test cases that have detected faults before
    - They probably execute code with a high fault density
  - Structural: Priority for executing elements that have not been recently executed
    - Can be coarse-grained: Features, methods, files, ...

Summary

- Regression testing is an essential phase of software product development.
- In a situation where test resources are limited and deadlines are to be met, execution of all tests might not be feasible.
- One can make use of different techniques for selecting a subset of all tests to reduce the time and cost for regression testing.