

CITS5501 Software Testing and Quality Assurance

Graph-based testing

Unit coordinator: Arran Stewart

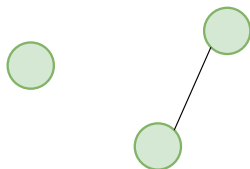
Overview

- Graph-based testing – we identify inputs which will exercise particular *paths* through a graph representing the software in some way.
- The graph could represent
 - control flow through a program
 - data flow between variables
 - an activity diagram, showing the workflow when a user interacts with the system
 - a state diagram, showing states of a system and transitions between them

Graph definition

A graph consists of:

- A set N of nodes
- A set E of edges, each edge being an “arrow” from one node to another



Graph-based testing

We will start by considering control flow. Our approach is:

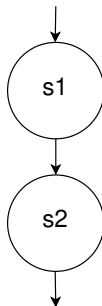
- ① Use the source code (or pseudocode) to produce a control flow graph.
- ② Using the graph produce a set of tests for the given program.

Constructing the graph

- In a control flow graph, nodes represent points in the program control flow can go “from” or “to”
- Loops, thrown exceptions and gotos (in languages that have them) are locations control flow can go *from* – statements representing these spots are “sources”
- Locations control flow can go *to* are “sinks”

Sequence control flow graphs

- The flow graph for a sequence of statements “s1; s2;” is

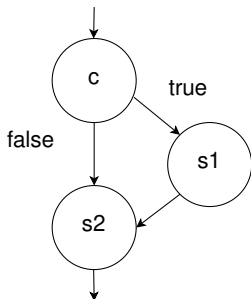


if-then control flow graphs

- given pseudocode like

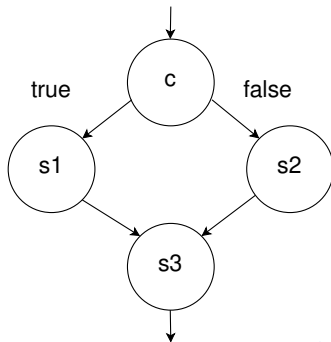
```
if c then:  
  s1  
s2
```

we get the following graph



if-then-else control flow graphs

- if c then:
 s1
else:
 s2
s3

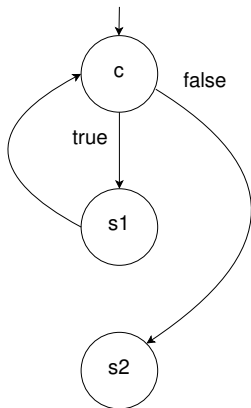


What about loops?

- Edges will obviously go “backward” in the graph (usually, towards the “top”)

while-do control flow graphs

```
• while c do:  
  s1  
  s2
```



other structures

- Most other control flow structures can be written into one of these forms (including “case” statements, breaking out of loops, “for” loops, etc)

other structures – example

A “case” statement:

```
case x of:  
  val1: s1; break  
  val2: s2; break  
  default: s3  
s4
```

Can be written as nested if-else

```
if x == v1:  
  s1  
else:  
  if x == v2:  
    s2  
  else:  
    s3  
s4
```

Using the graph

- To find a new test, examine the graph edges that *haven't* been exercised yet, and try to devise a test that exercises it
- In general, we'd actually like to find a test that exercises as *few* edges as possible
- why?

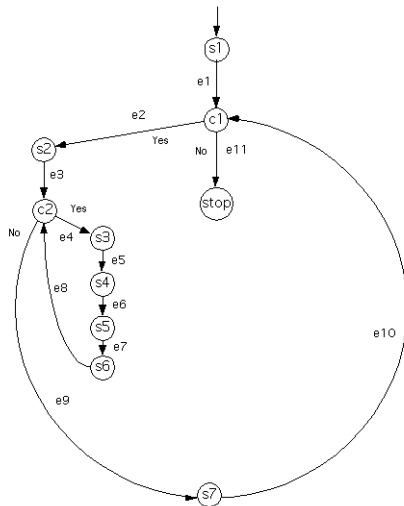
Using the graph

- To find a new test, examine the graph edges that *haven't* been exercised yet, and try to devise a test that exercises it
- In general, we'd actually like to find a test that exercises as *few* edges as possible
- why?
 - Tests that exercise a large number of edges usually represent “common” scenarios – we'd actually like to find less common cases (i.e. get more “value” out of the test)
 - Ideally, we want tests to be small and independent, so that when something goes wrong, we can localize the fault.

Example – sorting algorithm

```
S1  i = 2
C1  while (i <= n):
S2      j = i - 1
C2      while j >= 1 and A[j] >= A[j+1]:
S3          temp = A[j]
S4          A[j] = A[j+1]
S5          A[j+1] = temp
S6          j = j-1
S7      i = i + 1
```

Example – sorting algorithm (2)



Example – binary search

Inputs

- n , the length of the following array.
- A , an integer array with entries $A[1], \dots, A[n]$ such that $A[i] < A[i+1]$ for i between 1 and $n-1$ (i.e., it's sorted in ascending order, and 1-based)
- key , an integer to search for (the “needle”)

Outputs

- $index$, an integer between 0 and n such that:
 - if $index = 0$ then key does not equal any entry of the array A
 - if $index$ is between 1 and n then $A[index] = key$

Example – binary search (2)

```
found = false
low = 1
high = n
while ((low <= high) and not found):
    medium = floor((low + high)/2)
    if A[medium] == key:
        index = medium
        found = true
    else:
        if A[medium] < key then
            low = medium + 1
        else:
            high = medium - 1

if not found:
    index := 0
```

Graph-based testing criteria

Graph-based testing criteria

- Some possible criteria include:
 - node coverage - our test set traverses every node
(if using program control flow: statement coverage is similar, but coarser)
 - edge coverage - we traverse every edge
 - edge-pair coverage - we traverse every possible *pair* of edges
- We might use the informal heuristic of executing each loop 0 times, once, more than once (sometimes called “loop coverage”)

Prime paths

Definitions:

- **Simple path:** A path from node n_i to n_j is simple if no node appears more than once, except possibly the first and last nodes are the same
 - No *internal* loops in our path
 - A loop is a simple path

Prime paths

Definitions:

- **Simple path:** A path from node n_i to n_j is simple if no node appears more than once, except possibly the first and last nodes are the same
 - No *internal* loops in our path
 - A loop is a simple path
- **Prime path:** A simple path that does not appear as a proper subpath of any other simple path

Prime path coverage

- **Prime Path Coverage (PPC)**: Every prime path in the graph is visited.
- It subsumes node and edge coverage
- But not edge-pair coverage – we code have nodes (m,n) , where m loops to itself, and edge pair coverage requires the path (m,m,n) to be exercised.
- when it comes to devising *tests*, some tests may end up exercising multiple prime paths. But that's okay – as long as all prime paths are visited, we've satisfied the criterion.

Control flow graphs

In a control flow graph, different graph coverage criteria will correspond to:

- Node coverage: Execute every statement
(in fact, node coverage is stronger, since one statement may expand to multiple nodes)
- Edge coverage: Execute every branch

Note that complex boolean conditions in branches are still treated as a single node. (Effectively, the boolean condition is a “black box”.)

Logic coverage conditions (used, for instance, in avionics) look at these conditions in finer-grained detail.