SOFTWARE TESTING: TUTORIAL 2 DISCUSSION

In tutorial 2 you were given this piece of code and asked to generate three test suites which respectively satisifed statement, branch and basic condition coverage criteria:

Original code under test

```
vector doGraham(Vector p) {
  int i,j,min,M;

  Point t;
  min = 0;

  // search for minimum:
  for(i=1; i < p.size(); ++i) {
    if( ((Point) p.get(i)).y <
        ((Point) p.get(min)).y )
    {
      min = i;
    }
  }

  // continue along the values with same y component
  for(i=0; i < p.size(); ++i) {
    if(( ((Point) p.get(i)).y ==
          ((Point) p.get(min)).y ) &&
        (((Point) p.get(i)).x >
          ((Point) p.get(min)).x ))
    {
      min = i;
    }
  }
```

Here are some sample answers:

Test suites for various coverage criteria		
Coverage criterion	**Test vectors: { [x,y]* }+**	**Expected results**
Statement	{ [0,1], [1,0], [2,0] }	*min = 2*
Branch	{ [1,1], [-1,-1] }	*min = 1*
	{ [-1,-1], [2,-1] }	*min = 1*
Basic condition	{ [6,5], [4,4], [5,4] }	*min = 2*

Note that the branch coverage test suite comprises two successive calls to `doGraham` with different test vectors. It's also worth noting that *all three* of the above test suites actually satisfy all three criteria! This is a strong argument in favour of developing your tests incrementally: start with something simple, see if it satisfies the adequacy criterion that you're aiming for, and if it doesn't then observe what statements/branches/conditions were missed and target them next.

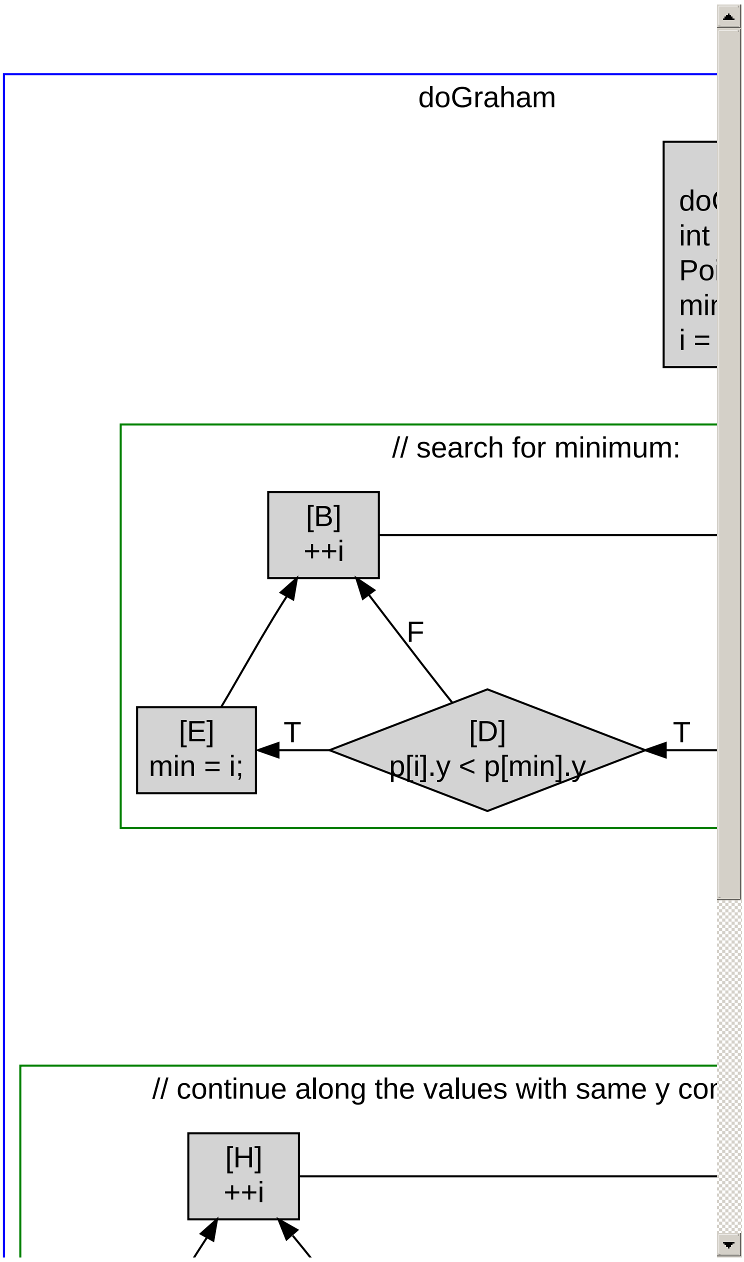
Activity five suggests that you take these test suites and generate three mutants of the original code (one for each coverage criterion) which **are not killed** by their respective suite. You need to be very careful here that you create mutants which **aren't equivalent to the original program**. It's best in this situation to also try to come up with a test that shows that your mutant does break the code (i.e. a test that “kills” the mutant, to use mutation testing terminology — more about that in Lecture 9…).

The thing to draw from this activity is that a “perfect” test suite should reject (i.e. fail) all programs which aren't correct. In practice though, they don't. Clearly here, your tests can obtain 100% branch coverage (and other measures) and still not detect many broken versions of the correct program.

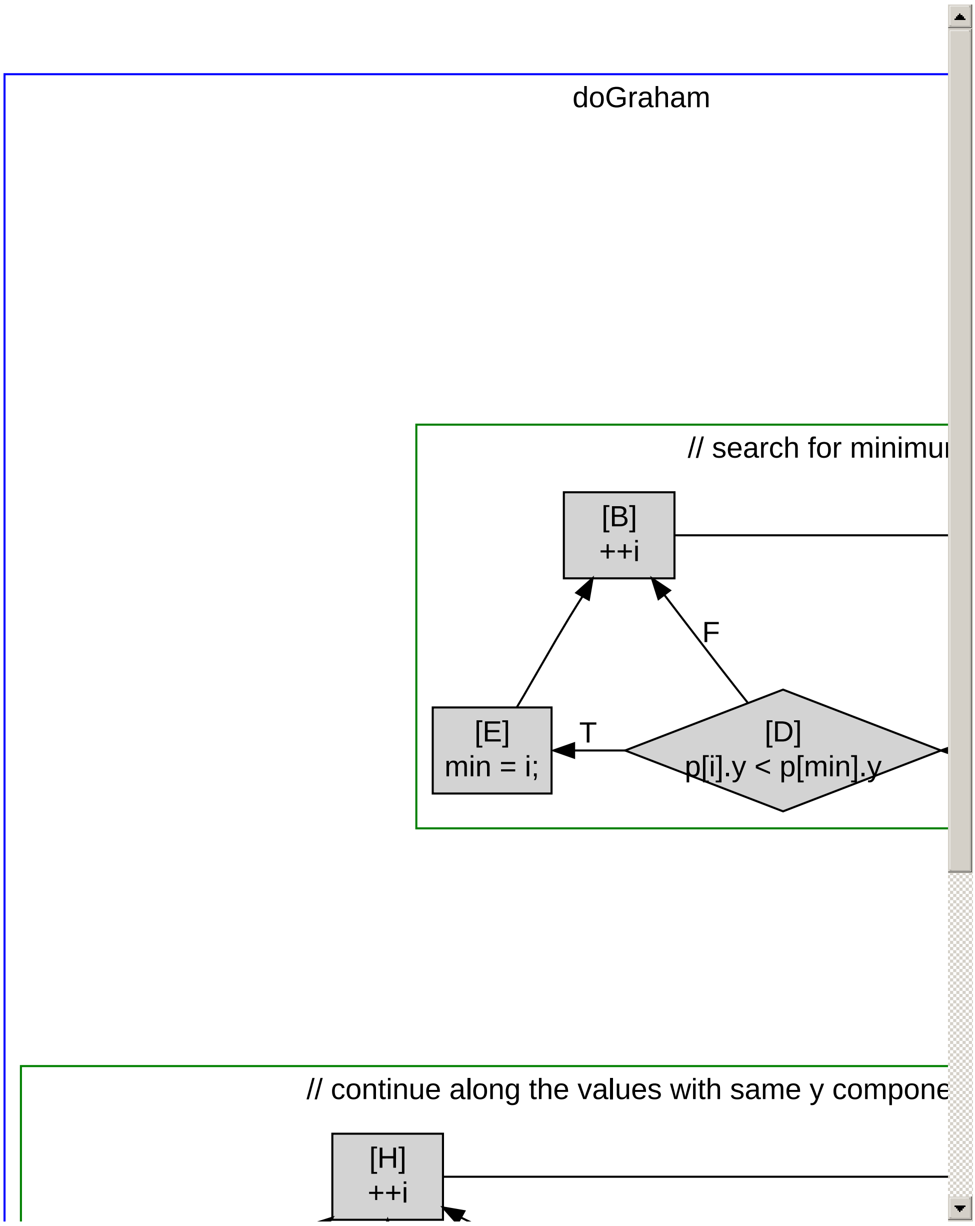
Here's a workthrough for each criterion and test above:

CONTROL FLOW GRAPH

Here are two control flow graphs for the code. The first will be useful for developing statement/branch coverage:



The second breaks the compound condition in the second loop apart, and will be helpful for developing condition-based coverage since it makes Java's short-circuiting behaviour clearer (node L becomes two nodes, L and M).



Note how for loops are handled in both graphs: the initialiser (i = 1 or i = 0) is executed before the loop; then the loop condition is evaluated; then the loop body is executed; and finally the counting expression (++i) is executed before returning to the top of the loop.

Also note how in nodes D, L, and M I use a shorthand for the rather long-winded Java expressions in the source. The annotations you use on a graph are there to make it clear to you what's happening, so you don't need a literal copy of the entire source code — but something which makes it clear what's happening will be helpful.

STATEMENT COVERAGE

Statement-adequate test suite	
Test vector	**Expected result**
{ [0,1], [1,0], [2,0] }	*min = 2*

A good place to start with this kind of problem is loops — modify the initialisation, termination condition or increment in order to iterate too many or too few times. So changing i=1 to i=0 on line 8 looks like a good start:

Mutant SC1:

```
7:  // search for minimum:
8:  for(i=0; i < p.size(); ++i) {
9:    if( ((Point) p.get(i)).y <
```

This mutant is not killed by any of the tests, so it looks promising. Unfortunately however, **it can't be killed by any test at all**, or at least not any test which just pays attention to the program's output based on valid input vectors: *min* starts off at 0, so Mutant SC1 just compares the first vector element with itself. Unnecessary, but harmless. You couldn't even detect this mutant by (say) supplying a vector with a single null element, as while Mutant SC1 would throw an exception at line 9, Original would also throw an exception — just this time at line 18. To identify this mutant, your test harness would have to be counting the number of accesses to p.get(), or something similar. There *are* circumstances in testing of a system where you might do this, but all we're doing is looking at the final value of *min*, so for our purposes **Mutant SC1 is equivalent to Original**!

All of the above tests result in *min* ≥ *1*, so changing `i=0` to `i=1` on line 17 looks like a good next try:

Mutant SC2:

```
16:  // continue along the values with same y component
17:  for(i=1; i < p.size(); ++i) {
18:    if(( ((Point) p.get(i)).y ==
```

However this is also an unfortunate choice! The first loop finds the first point in the vector with the minimum *y* value, and the second loop is there in order to search through *all* points with this minimum *y* and find the one which has the maximum *x* value. So the first comparison made in the loop (with *i=0*) is actually always unnecessary — in fact the second loop could start with `i=min+1` since *min* will always be the first point with the minimum *y*, and the second loop should only be looking at subsequent points *min + 1*, *min + 2*, … So **Mutant SC2 is equivalent to Original** too!

One of the unfortunate aspects of doing a course in Software Testing is that we spend a depressing amount of time looking at really awful code: in this case for example, it's trivial to achieve what these two loops do in a single loop. It's not *really* awful, but it could be simultaneously clearer, more concise and more efficient.

This is one of the big problems with mutation testing: a huge number of mutants are actually equivalent to the original code. The change they involve has no effect on program execution. This is a waste, and in many cases unfortunately unavoidable: think how smart a mutant generator would have had to have been in order to recognise equivalence of the above three programs.

Righty. Let's get us a proper distinct mutant. Let's try `i=2`:

Mutant SC3:

```
16:  // continue along the values with same y component
17:  for(i=2; i < p.size(); ++i) {
18:    if(( ((Point) p.get(i)).y ==
```

Good-oh. This mutant still gives *min=2* on our original statement coverage test “suite”. In order to kill it now we need to add a test whose *min* is less than 2 and is found by the second loop. This, for example, would do:

Mutant SC3-killing test		
Test vector	**Expected result**	**SC3 result**
{ [0,1], [1,1] }	*min = 1*	*min = 0*

Note that the new test on its own doesn't achieve statement coverage (it never executes line 12), so it's worth keeping the old test around:

Mutant SC3-killing statement-adequate test suite		
Test vector	**Expected result**	**SC3 result**
{ [0,1], [1,0], [2,0] }	*min = 2*	
{ [0,1], [1,1] }	*min = 1*	*min = 0*

BRANCH COVERAGE

Branch-adequate test suite	
Test vector	**Expected result**
{ [1,1], [-1,-1] }	*min = 1*
{ [-1,-1], [2,-1] }	*min = 1*

Here we observe that both of our tests give a result *min = 1*. So how about a mutant which gives a result of 1 a little more often than it should? Let's replace one of the `min = i` statements with `min = 1`. This kind of typo is particularly hard to spot, and a strong reason why variable names like *i* and *l* should be avoided:

Mutant BC1:

```
22:    {
23:      min = 1;
24:    }
```

This mutant would be killed by any test whose result should be a *min > 2*, and where that point shares a *y* with an earlier point:

Mutant BC1-killing test		
Test vector	**Expected result**	**BC1 result**
{ [-1,-1], [2,-1], [3,-1] }	*min = 2*	*min = 1*

An alternate trick here would be to set `min = 1` initially:

Mutant BC2:

```
4:   Point t;
5:   min = 1;
6:
```

This mutant will go wrong on vectors consisting of only one point (with an out of bounds exception), and inputs where *min* should end up 0, but that point shares a *y* value with another — either of these tests will kill it:

Mutant BC2-killing tests		
Test vector	**Expected result**	**BC2 result**
{ [-1,-1] }	*min = 0*	Exception
{ [-1,-1], [-2,-1] }	*min = 0*	*min = 1*

Finally, just to show that constants aren't the only good sources of mutants we could change the `&&` on line 19 to `||`:

Mutant BC3:

```
18:   if(( ((Point) p.get(i)).y ==
19:         ((Point) p.get(min)).y ) ||
20:      (((Point) p.get(i)).x >
```

This mutant will go wrong in all kinds of situations, but not on our current test suite. Here's a killer:

Mutant BC3-killing test		
Test vector	**Expected result**	**BC3 result**
{ [1,3], [0,3] }	*min = 0*	*min = 1*

Don't forget to include enough of the original test suite that you still get branch adequacy:

Mutant BC1-killing branch-adequate test suite		
Test vector	**Expected result**	**BC1 result**
{ [1,1], [-1,-1] }	*min = 1*	
{ [-1,-1], [2,-1], [3,-1] }	*min = 2*	*min = 1*

Mutant BC2-killing branch-adequate test suite		
Test vector	**Expected result**	**BC2 result**
{ [1,1], [-1,-1] }	*min = 1*	
{ [-1,-1], [2,-1] }	*min = 1*	
{ [-1,-1] }	*min = 0*	*min = 1*

Mutant BC3-killing branch-adequate test suite		
Test vector	**Expected result**	**BC3 result**
{ [1,1], [-1,-1] }	*min = 1*	
{ [-1,-1], [2,-1] }	*min = 1*	
{ [1,3], [0,3] }	*min = 0*	*min = 1*

BASIC CONDITION COVERAGE

Basic condition-adequate test suite	
Test vector	**Expected result**
{ [6,5], [4,4], [5,4] }	*min = 2*

This test suite also fails to kill Mutant SC3, so we can re-use Mutant SC3 here, and use the same test from the statement-adequate suite to kill it:

Mutant SC3-killing basic condition-adequate test suite		
Test vector	**Expected result**	**SC3 result**
{ [6,5], [4,4], [5,4] }	*min = 2*	
{ [0,1], [1,1] }	*min = 1*	*min = 0*

CODA

One tutorial group came up with this comprehensive test suite to achieve basic condition coverage:

Test suites for various coverage criteria		
Coverage criterion	**Test vectors: { [x,y]* }+**	**Expected results**
Basic condition	{ }	*min = 0*
	{ [0,0], [1,2] }	*min = 0*
	{ [1,2], [1,1] }	*min = 1*
	{ [5,0], [2,1], [6,0] }	*min = 2*

This is good in that it exercises the code more thoroughly (notice that loops are covered for a number of different iteration counts, including zero). However it's pretty scary when it comes to generating a mutant that can sneak past it. My suggestion here is based on the observation that the resulting values for *min* here are all quite small: if we change the type of the index variables from `int` to `byte` (in the first line of the method), then their maximum value becomes 127.

Evil Mutant:

```
1: vector doGraham(Vector p) {
2:   byte i,j,min,M;
3:
```

Supplying a test vector with more than 128 entries will expose this fault if the lowest-y/highest-x point is beyond index 127 in the vector.

Version 1.1, 2018/01/09 13:42:03

Informatics Forum, 10 Crichton Street, Edinburgh, EH8 9AB, Scotland, UK
Tel: +44 131 651 5661, Fax: +44 131 651 1426, E-mail: school-office@inf.ed.ac.uk