The University of Melbourne
SWEN90006: Software Testing and Reliability

Tutorial 5 Solutions

Semester 2, 2017

Introduction

Encapsulation is an abstraction mechanism that aids in programming, but that adds complexity to testing. We often need to break the information hiding utilised by classes in order to examine the class state for the testing purposes.

The aim of this tutorial is for you to explore some of the issues in object oriented testing through the simple Graph given below. The graph uses an adjacency matrix that records which vertices are "*adjacent*" in the graph.

Task 1

Consider the `addEdge` and `deleteEdge` methods in the `Graph` class below. Derive test cases for path coverage and condition coverage for these two methods. **Note** that it may be necessary to examine the state variables in your test cases. Sketch how you would achieve this.

Answer

As always we start with the control-flow graphs for `addEdge` and `deleteEdge`. We give the control-flow graph for the `deleteEdge` operation in Figure 1. The `addEdge` operation is similar.

There are two paths in the control-flow graph and two conditions. To test the `deleteEdge` however, we do require that `Graph` objects be in the right state for testing. The input domain is:

$$\texttt{int} \times \texttt{int} \times \texttt{int[]} \times \texttt{int[][]} \times \texttt{int} \times \texttt{int}$$

from the state of the object and the parameters of the function:

$$\texttt{\_order} \times \texttt{\_allocated} \times \texttt{\_vertices} \times \texttt{\_matrix} \times \texttt{m} \times \texttt{n}.$$

Note that the `Graph` class is not sufficiently complete as there are no observers in the set of operations. Consequently, we would need to write some functions to observe the `Graph` state.

For path coverage we will need to have `mIndex < _order && nIndex < _order` true and false. A test case for executing path ABC is given in Figure 2 and with the input parameters

$$\texttt{deleteEdge}(5, 6).$$

Note that we have already tested both conditions in the branch at node C to be true. Now consider a test case for ABC. Consider the same graph as in Figure 2 above and make a call to

$$\texttt{deleteEdge}(8, 9).$$

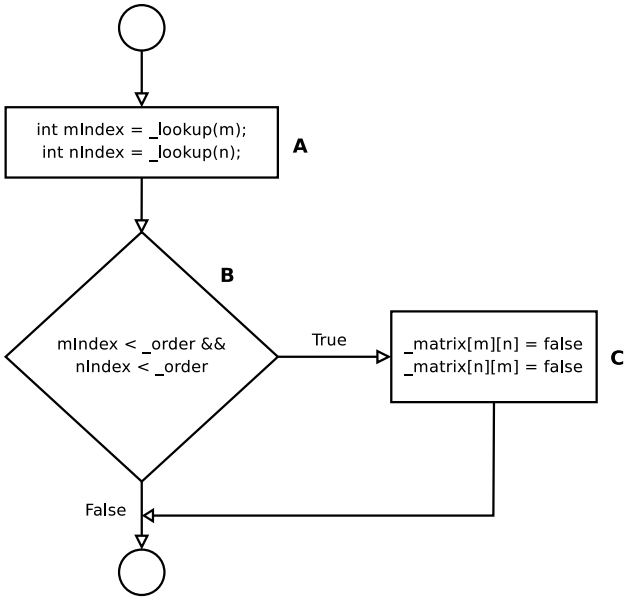

Figure 1: The control-flow graph for the `deleteEdge` operation.

```
_order = 6
_allocated = 6
_vertices[] = {1, 2, 3, 4, 5, 6}
```

The variable `_matrix[][]` is defined as below.

	1	2	3	4	5	6
1		T	T			
2	T			T	T	
3	T				T	
4		T				T
5		T	T			T
6				T	T	

The blank entries are assumed to be false. The expected output would have the same state variables except that `_matrix` would be changed to be:

	1	2	3	4	5	6
1		T	T			
2	T			T	T	
3	T				T	
4		T				T
5		T	T			
6				T		

Figure 2: The state of a `Graph` object.

The function `_lookup` returns `_order+1` for the value of `nIndex` and so the second condition is false and we take path AB (but what will the program do on this path?)

Task 2

Draw a finite state automaton for `Graph` class. **Note** that it is not always possible to add an edge and it is not always possible to delete a vertex. You will need to consider the states and the guards on transitions to ensure all of the conditions.

Answer

To draw a state transition diagram, we must first determine what states we want to include. We partition the state using input partitioning (recall that a state variable is an input that is passed implicitly) and then select boundary cases. We identify 7 states for this class. Assuming that V represents the set of vertices, and E represents the set of edges, then the 7 states are:

$$
\begin{array}{ll}
1 & \#V = 0, \#E = 0 \\
2 & \#V = 1, \#E = 0 \\
3 & 1 < \#V < K, \#E = 0 \\
4 & 1 < \#V < K, \#E > 0 \\
5 & \#V = K, \#E = 0 \\
6 & \#V = K, 0 < \#E < K(K-1)/2 \\
7 & \#V = K, \#E = K(K-1)/2
\end{array}
$$

In the above, $\#V$ represents the size of the set of vertices (the number of vertices), and similarly for edges. The total number of possible edges is restricted by the number of vertices. If V is the number of vertices, then $V(V-1)/2$ is the total number of edges that are possible. The expression $\#V(\#V-1)/2$ represents the number of edges in the graph if we were to draw a link between every pair of vertices in the graph. K represents the maximum number of vertices (passed to the class constructor).

Now that we have identified the states that we wish to test, we create transitions between these states whenever it is possible. The resulting FSA for this is shown in Figure 3.

Note that in this figure, there are several transitions missing. At every state, there should be a transition that loops back to that state that covers the following cases:

- Adding a vertex/edge that is already in the graph;
- Deleting a vertex/edge that is not in the graph;
- Adding an edge in which the vertices that make up this edge are not in the graph; and
- Deleting a vertex that is part of a edge.

These cases result in the state of the graph remaining unchanged. To improve the readability of the graph, they are omitted.

The 7 states above may be considered overkill, and you may wish to collapse some of them into one state. This reduces the number of transitions as well.

Task 3

Derive a set of test cases to test every transition in your graph.

Answer

To derive the test cases, we need to derive the test sequences from the transitions, and test the state of the module after every transition.

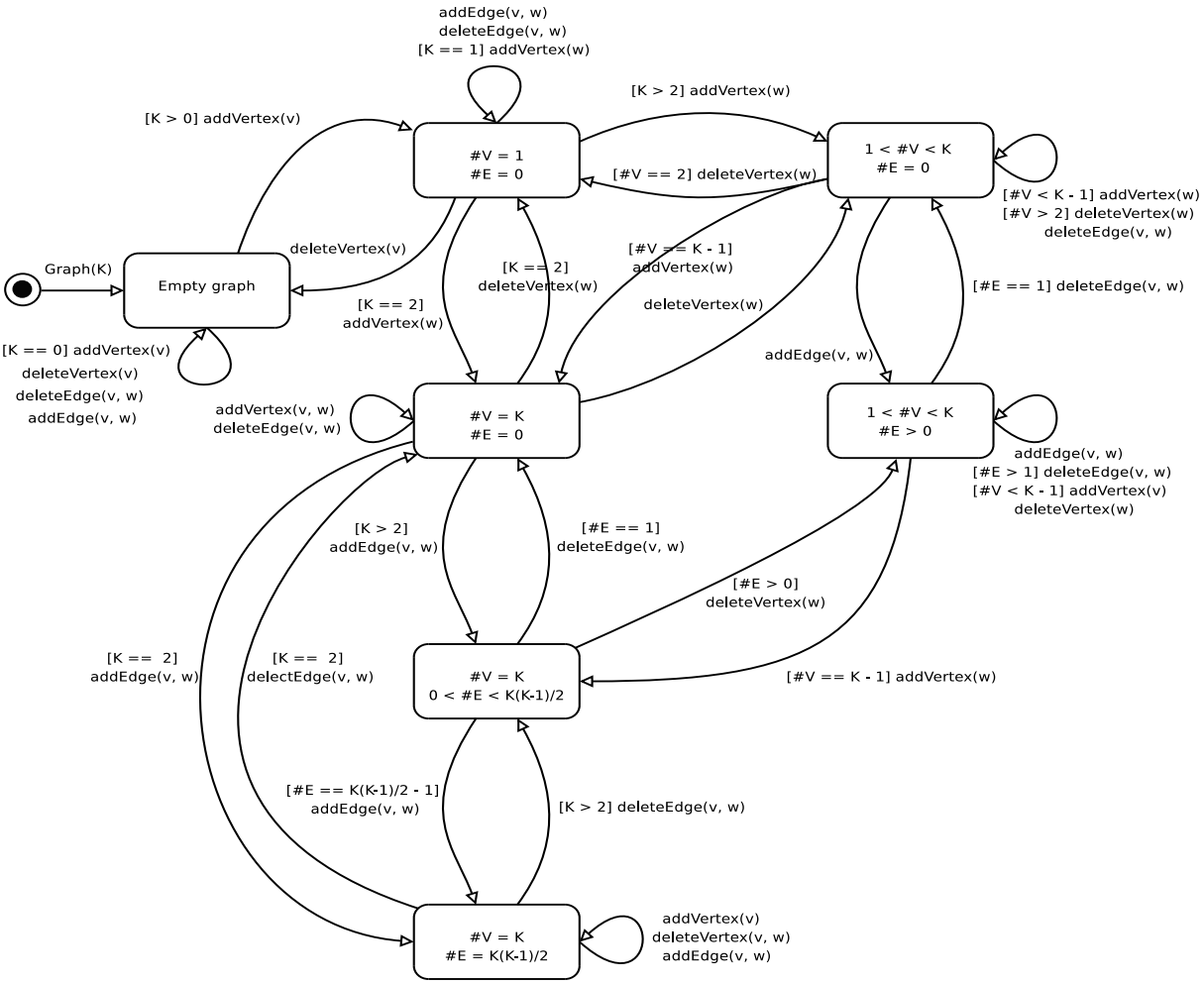

Figure 3: The State Transition Diagram including states and guards for the Graph class

In this solution, we consider only the case in which the maximum size of the graph is 2. To move between states, the following sequence achieves transition covered for the cases in which $K == 2$:

`Graph(2); addVertex(1); addVertex(2); addEdge(1,2); deleteEdge(1,2); deleteVertex(2); deleteVertex(1)`

In fact, we need to extend this sequence to test transitions that loop from a state to itself, however, we omit this for now.

Between each call, we need to check whether the state of the object that we are testing has the correct value. This is difficult because there are no observer methods in the `Graph` class. To observe the state value, we must augment the implementation. Thus, we add two method called `getVertices()` and `getEdges()`, which return the vertices and edges as instances of the `Set` class, found in the Java library.

Using this new method, we would traverse the test sequence above, and after the first transition, `Graph(2)`, we would test that the vertices and edges sets are empty:

```
Graph g = new Graph(2);
Set emptySet = createSet();

assert g.getVertices().equals(emptySet);
assert g.getEdges().equals(emptySet);
```

After adding two vertices and an edge, we would the state again. To test that 2 is the maximum size, we would attempt to add another vertex (traversing the `addEdge` transition in the FSA), and check the state again. If the state changes, then we raise a failure.

To traverse every transition in the FSA, we would also require cases in which $K == 0$, $K == 1$, and $K > 2$.

Question 4

Using 5 different mutant operators from Chapter 6 of the lecture notes, derive 5 mutants for the `addEdge` operation. Do the test cases derived from your finite state automaton kill all of the mutants? If not, are the live mutants equivalent, or are the test cases not yet adequate?

The Graph Abstract Data Type

The Graph Interface

```
class Graph
{
    public Graph(int n);
      // The constructor for graphs. It initialises
      // the graph state and sets up the graph
      // representation for graphs of n or less nodes.

    public void addVertex(int v);
      // Add a vertex to the vertex graph.

    public void addEdge(int m, int n)
      // Add an edge to the graph. Edges are specified by pairs
      // of vertices. To add the edge correctly it is necessary
      // that m and n have already been added to graph as vertices.

    public void deleteVertex(int v);
      // We can only delete a node if it is not part of some edge
      // in the graph and it exists as an actual vertex in the
      // graph.

    public void deleteEdge(int m, int n);
      // We can only delete an edge if the two specified
      // vertices, m and n, are in the graph.
}
```

The Graph State and Hidden Functions

```
//------------ Private Attributes ----------------------------
//    The representation of a graph consists of an array
//    of vertices that map nodes (just integers) to array
//    indexes. The adjacency matrix _matrix sets matrix[i][j]
//    to true if there is an edge between _vertices[i] and
//    _vertices[j].
//
//    The operations must maintain the following invariant:
//        _vertices[i] is defined iff i < _allocated
//        _allocated <= _order

private int     _order;      // The number of vertices allowed
private int     _allocated;  // The next free space in the vertex array
private int     _vertices[]; // A list of the actual vertices
private boolean _matrix[][]; // The adjacency matrix

//------------- Private Methods -------------------------------

private static boolean[][] _allocate(int n)
{
  return new boolean[n][n];
}

private static int[] _vertices(int n)
{
  return new int[n];
}

private int _lookup(int m)
{
  int index = 0;
  while (index < _allocated && !(_vertices[index] == m))
    index = index + 1;

  if (index == _allocated) {
    return _order + 1;
  }
  else {
    return index;
  }
}
```

Adding and Deleting Edges

```
public void addEdge(int m, int n)
{
  // Add an edge to the graph. Edges are specified by pairs
  // of vertices. To add the edge correctly it is necessary
  // that m and n have already been added to graph as vertices.

  int mIndex = _lookup(m);
  int nIndex = _lookup(n);

  if (mIndex < _order && nIndex < _order)
  {
    _matrix[mIndex][nIndex] = true;
    _matrix[nIndex][mIndex] = true;
  }
}

public void deleteEdge(int m, int n)
{
  // We can only delete an edge if the two specified
  // vertices are in the graph.

  int mIndex = _lookup(m);
  int nIndex = _lookup(n);

  if (mIndex < _order && nIndex < _order) {
    _matrix[mIndex][nIndex] = false;
  }
}
```