ESOF 422 - Homework 5

Nathan Stouffer Kevin Browder

March 11, 2020

Question 1

How else could we compare test criteria besides subsumption?

Percentage of complete path coverage is another way of comparing test criteria. First we would need to determine the total number of paths for complete path coverage.

For each node in the graph, we would run a depth first search. Every time a path is found we add 1 a running total. After this is completed we will have the total number of paths in a graph.
Then for each criteria we can determine all paths needed to satisfy the criteria. We then simply divide the paths in the criteria by the total number of paths in the graph to get a percentage of the total paths covered by a test criteria.

The larger the percentage the better the test criteria. This could be used to compare test criteria.

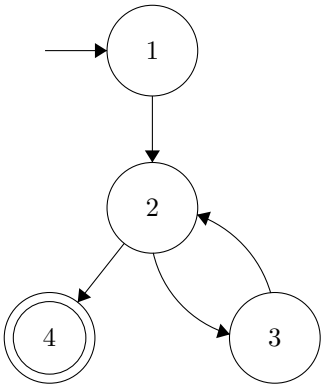
Question 2

Book Question 4

We consider the following graph

$N = \{1, 2, 3, 4\}$
$N_0 = \{1\}$
$N_f = \{4\}$
$E = \{(1, 2), (2, 3), (3, 2), (2, 4)\}$

(a) We depict the graph below



(b) There are no test paths that achieve Node Coverage but not Edge Coverage. For the above graph, any path that satisfies Node Coverage must also satisfy Edge Coverage. This is because the graph has no branches to follow (only a cycle). For example, the test path [1, 2, 3, 2, 4] satisfies NC and EC.

(c) A test path that satisfies EC but not EPC is [1, 2, 3, 2, 4]. This does not achieve EPC because it is missing [3, 2, 3]

(d) The set of test paths that satisfy EPC is $\{[1, 2, 3, 2, 3, 2, 4], [1, 2, 4]\}$

Book Question 5

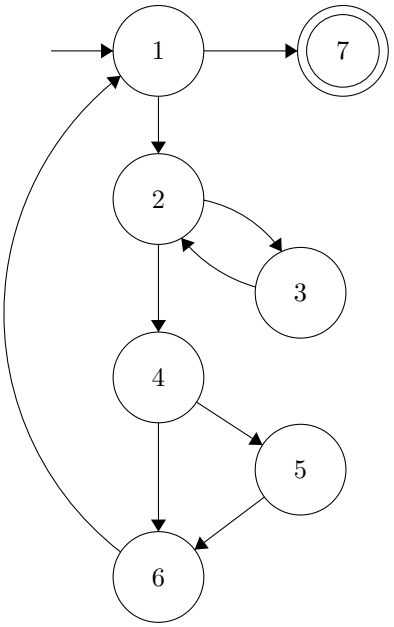
We now consider the following graph

$N = \{1, 2, 3, 4, 5, 6, 7\}$
$N_0 = \{1\}$
$N_f = \{7\}$
$E = \{(1, 2), (1, 7), (2, 3), (2, 4), (3, 2), (4, 5), (4, 6), (5, 6), (6, 1)\}$

We also consider the following (candidate) test paths

$p_1 = [1, 2, 4, 5, 6, 1, 7]$
$p_2 = [1, 2, 3, 2, 4, 6, 1, 7]$
$p_3 = [1, 2, 3, 2, 4, 5, 6, 1, 7]$

(a) We depict the graph below



(b) We now list the test requirements for EPC on the above graph. TR = { [1, 2, 3], [1, 2, 4], [2, 3, 2], [2, 4, 5], [2, 4, 6], [3, 2, 3], [3, 2, 4], [4, 5, 6], [4, 6, 1], [5, 6, 1], [6, 1, 2], [6, 1, 7] }

(c) The given set of test paths (p_1, p_2, p_3) do not satisfy Edge-Pair Coverage. They are missing [3, 2, 3] and [6, 2, 1] so they do not satisfy EPC.

(d) This question considers the simple path [3, 2, 4, 5, 6] and the test path [1, 2, 3, 2, 4, 6, 1, 2, 4, 5, 6, 1, 7]. For the given paths, the test path does not tour the simple path directly. It takes a side trip through the path [6, 1, 2].

(e) We now give the test requirements for Node Coverage, Edge Coverage, and Prime Path Coverage.

TR_{NC} = { [1], [2], [3], [4], [5], [6], [7] }

TR_{EC} = { [1, 2], [1, 7], [2, 3], [2, 4], [3, 2], [4, 5], [4, 6], [5, 6], [6, 1] }

TR_{PPC} = { [3, 2, 4, 5, 6, 1, 7], [1, 2, 4, 5, 6 ,1], [2, 4, 5, 6, 1, 2], [3, 2, 4, 6, 1, 7], [4, 5, 6, 1, 2, 3], [4, 5, 6, 1, 2, 4], [5, 6, 1, 2, 4, 5], [6, 1, 2, 4, 5, 6], [6, 1, 2, 4, 6], [1, 2, 4, 6, 1], [2, 4, 6, 1, 2], [4, 6, 1, 2, 3], [4, 6, 1, 2, 4], [2, 3, 2], [3, 2, 3] }

(f) From $\{p_1, p_2, p_3\}$, the test path $p_3 = [1, 2, 3, 2, 4, 5, 6, 1, 7]$ satisfies Node Coverage but not Edge Coverage.

(g) From $\{p_1, p_2, p_3\}$, the test paths $p_1 = [1, 2, 4, 5, 6, 1, 7]$ and $p_2 = [1, 2, 3, 2, 4, 6, 1, 7]$ achieve Edge Coverage but not Prime Path Coverage.

Question 3

Question 3 asks us to implement an algorithm that returns that prime paths of a graph. We do this with a Java implementation. Our program consists of two classes: Path and Graph.

The Path class is essentially just a list of nodes in the path. Operations that can be called on a path are queries that tell whether a given path visits a node, tours a subpath, or is a simple path.

The Graph class uses an adjacency matrix to represent a graph. A client of this class could compute all simple paths or all prime paths. We now describe the algorithm used to compute the prime paths of a graph.

To compute all prime paths, we first compute all simple paths. This is done by starting a depth first search at each node, with the extra condition that we only continue the search if the path is simple. Psuedocode is shown below where the variables "paths" is a global variable.

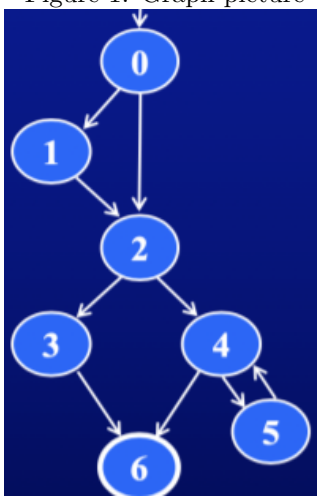
```
foreach node in Graph:
        path = node                 // start the path as the initial node
        dfsSimplePath(path)         // recursively compute all simple paths

dfsSimplePath(path):
        paths.add(path)                                 // add the path to a growing list
        foreach neighbor of path.last:                  // the last node in the path
                next = path.add(neighbor)               // create a new path
                if next.isSimple():                     // test if new path is simple
                        dfsSimplePath(next)             // keep computing dfs
```

Afte we generate all simple paths, we then sort the paths by length. We then know that the longest simple path is certainly prime. We then iterate through the list of simple paths (largest to smallest). At each iteration, we check if the simple path is a subpath of any prime path. If not, we know that the path is prime and we can add it to the list of prime paths. We then just return that list.

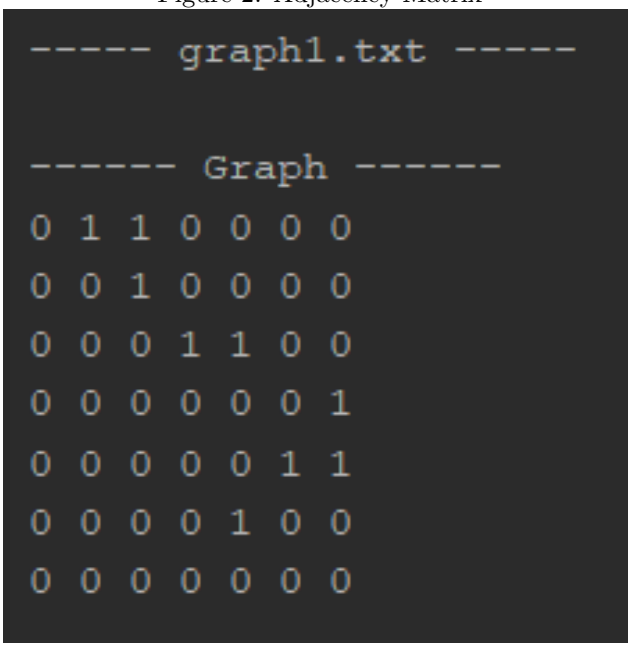
To test our program, we used graph shown below in Figure 1.

Figure 1: Graph picture



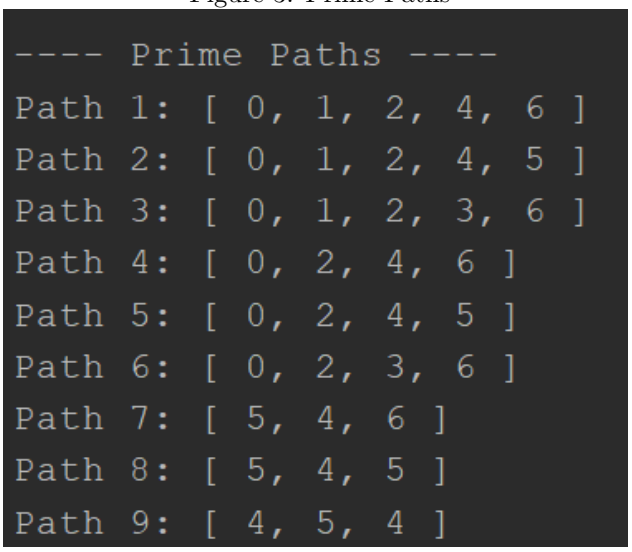
Below (in Figure 2), we show the adjacency matrix representing our graph.

Figure 2: Adjacency Matrix



The output of our program (all the prime paths) is shown below in Figure 3.

Figure 3: Prime Paths



In addition to this description and the figures with the output, we also submitted a our .java files and the .txt files with graph inputs.