


CSC263 PS3

Written by: Youngsin Ryoo, 

1 Question 1

1.1 a

Scenario 1

In this scenario we are looking for the number of unique paths from the start node to the end node, where you can visit each node only once. In the following algorithm, V, E are the list of vertices and edges in the graph respectively. Let $start \in V$ and $end \in V$ represent the start and end nodes in the G , respectively. Since there is no difference between cities and villages in this scenario, we can treat all non-port nodes equally. The parameter p is a set of nodes in the current path.

Algorithm 1

```
1: procedure GET_NUM_PATHS( $G = (V, E), u = start, p = \{start\}$ )
2:   num_paths  $\leftarrow$  0
3:   for each  $(u, v)$  in  $E$  do
4:     if  $v$  is  $end$  then
5:       num_paths  $\leftarrow$  num_paths + 1
6:     else if  $v$  is not in  $p$  then
7:       add( $p, v$ )
8:       num_paths  $\leftarrow$  num_paths + get_num_paths( $G, v, end, p$ )
9:       remove( $p, v$ )
10:    end if
11:  end for
12:  return num_paths
13: end procedure
```

Scenario 2

In this scenario we are looking for the number of unique paths from the start node to the end node, where you can visit each village node only once and a city node unlimited amount of times. $G = (V, E)$, $start \in V$ and $end \in V$ are defined exactly like in scenario 1. The parameter p is a multi-set of nodes in the current path.

Algorithm 2

```
1: procedure GET_NUM_PATHS( $G = (V, E), u = start, p = \{start\}$ )
2:   num_paths  $\leftarrow$  0
3:   for each  $(u, v)$  in  $E$  do
4:     if  $v$  is end then
5:       num_paths  $\leftarrow$  num_paths + 1
6:     else if  $v$  is not in  $p$  or is_city( $v$ ) then ▷ updated else if condition
7:       add( $p, v$ )
8:       num_paths  $\leftarrow$  num_paths + get_num_paths( $G, v, p$ )
9:       remove( $p, v$ )
10:    end if
11:  end for
12:  return num_paths
13: end procedure
```

Scenario 3

This scenario is exactly the same as scenario 2 except with the exception that a player can use a village passport card once on a village, which will allow them to visit the village one extra time. Let $W \subset V$ be the set of all villages. For every village $w \in W$ we will run a tweaked version of algorithm 2. This way we will collect the total number of paths from start to end with the special exception that the village w can be visited twice.

Algorithm 3

```
1: procedure SCENARIO3( $G = (V, E)$ )
2:    $W \leftarrow$  vertices that are villages
3:   total_num_paths  $\leftarrow$  0
4:   for each  $w$  in  $W$  do
5:      $p \leftarrow \{start\}$ 
6:     total_num_paths  $\leftarrow$  total_num_paths + get_num_paths( $G, start, p, w, 0$ )
7:   end for
8:    $p \leftarrow \{start\}$  ▷ case where passport is not used
9:   total_num_paths  $\leftarrow$  total_num_paths + get_num_paths( $G, start, p, nil, 0$ )
10:  return total_num_paths
11: end procedure
```

The parameter w holds the village (i.e vertex) that the player applies the passport card to. The parameter w_count holds the number of times village w shows up in the path p .

Algorithm 4

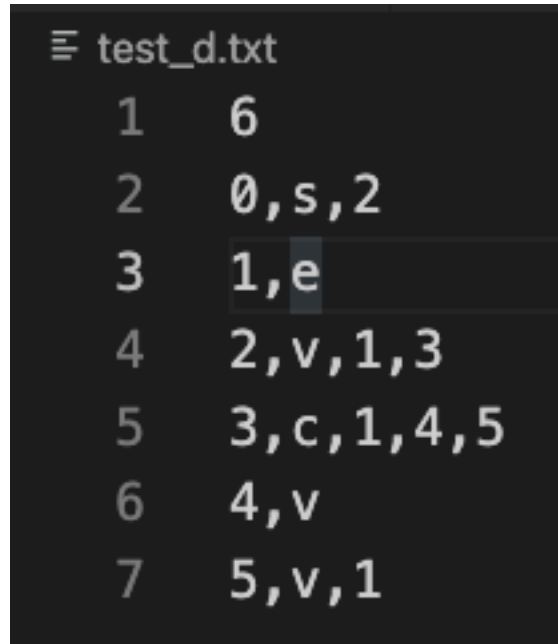
```
1: procedure GET_NUM_PATHS( $G = (V, E), u, p, w, w\_count$ )
2:   num_paths  $\leftarrow$  0
3:   for each  $(u, v)$  in  $E$  do
4:     if  $v$  is end then
5:       if  $w \neq \text{NIL}$  and  $w\_count < 2$  then
6:         continue ▷ Only count routes for which the village passport was used
7:       end if
8:       num_paths  $\leftarrow$  num_paths + 1
9:     else if  $v$  is not in  $p$  or is_city( $v$ ) or  $v$  is  $w$  then
10:      if  $v$  is  $w$  then
11:        if  $w\_count == 2$  then
12:          continue
13:        end if
14:         $w\_count \leftarrow w\_count + 1$ 
15:      end if
16:      add( $p, v$ )
17:      num_paths  $\leftarrow$  num_paths + get_num_paths( $G, v, \text{end}, p, w$ )
18:      remove( $p, v$ )
19:    end if
20:  end for
21:  return num_paths
22: end procedure
```

1.2 b

Our goal for representing the board is to have it be as concise as possible, yet capture are all the given scenarios. We store the following information:

1. The number of vertices in the graph (i.e the board).
2. For each vertex, it's type as one of village (v), city (c), start (s) or end (e), as well as the vertices it is adjacent to. Since the graph is undirected, it's sufficient to simply specify the adjacent vertex in just one of the vertex's adjacency list.

For simplicity, if there are n vertices in the graph, the vertex will correspond to a unique integer between 0 to $n - 1$. Here is the text file representation of the example board configuration:



```
≡ test_d.txt
1 6
2 0,s,2
3 1,e
4 2,v,1,3
5 3,c,1,4,5
6 4,v
7 5,v,1
```

The 6 at the start of the file represents the number of vertices in the graph. The numbers 0 through 5 uniquely represent the six vertices. Letters s, e, v and c represent the type of vertex and the comma separated numbers is the adjacency list for the corresponding vertex. Note that vertex 1 does not have any adjacent vertices in it's list but it is in the adjacency list of vertex 5. This is unconventional since usually both vertices would have the other vertex in it's adjacency list. However we decided to keep it this way to avoid redundancy.

1.3 c

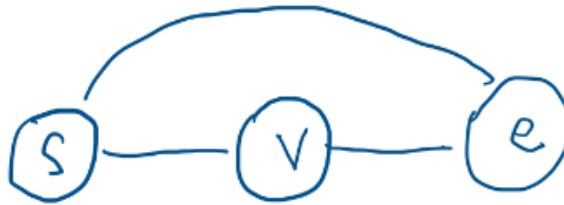
The code can be found in *Routes.py* file.

1.4 d

The test configurations have been carefully designed to satisfy the following statements:

1. The start node is not visited more than once.
2. Villages aren't visited more than once.
3. Routes are correctly computed for a simple case (i.e sanity check).
4. Cities are visited more than once when possible.
5. Usage of village passport is correctly reflected in the number of routes.

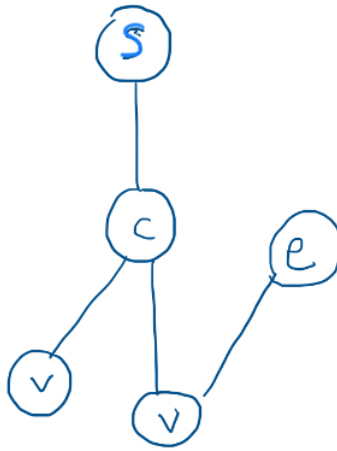
Test A



This is the simple case in which all the three scenarios have the same number of routes. This test cases is a simple sanity check to make sure there is nothing fundamentally wrong with the algorithm and implementation.

- expected scenario 1 routes: 2
- expected scenario 2 routes: 2
- expected scenario 3 routes: 2

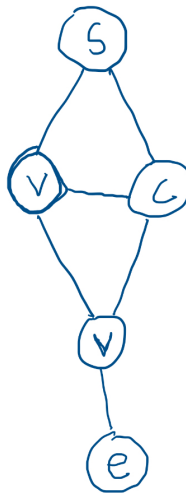
Test B



This case includes a dangling leaf with the parent node being a city. What this allows us to capture is the "back and forth" paths between a city and a village with a special passport applied to. The case helps us ensure statements 2, 4, 5.

- expected scenario 1 routes: 1
- expected scenario 2 routes: 2
- expected scenario 3 routes: 6

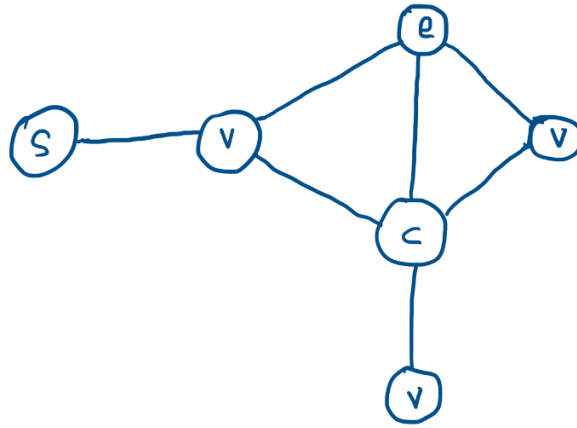
Test C



This test case captures multiple branches off the starting node. It also ensures statements 2, 3, 4, 5 holds true.

- expected scenario 1 routes: 4
- expected scenario 2 routes: 5
- expected scenario 3 routes: 18

Test D



The final test case is the example configuration board. This test case ensures that a complicated case with loops, leafs and multiple entries into the end works as expected. This case is important since it could potential capture edge cases that have been overlooked. This test case also helps us ensure statements 2, 3, 4, 5 holds true.

- expected scenario 1 routes: 3
- expected scenario 2 routes: 8
- expected scenario 3 routes: 39