Question 1

(a)

The BFS algorithm is capable of finding all paths in a graph starting at a given node. My proposed algorithm modifies BFS to able to store these paths and only extract the ones required by the specifications of the three scenarios.

The data structures used are the following:

- Dictionary representing the graph as an adjacency list. Keys are characters and values are arrays of characters.
- Queue storing arrays of characters.
- Array to store the current path.

We start with a path consisting of simply the 'S' character (which represents the starting point).
The queue starts with a single path array storing 'S'.

The rest of the algorithm loops while the elements in the queue are more than zero.
Inside the loop, pop the first element of the queue and assign it to `path`.
Store the last element of path in `last`.
If last equals 'F' (which represents the finish point), then the path is complete and can store it.
If it doesn't equal 'F' then we loop through all its adjacent edges. Notice that this is the same logic as with BFS but instead of having a queue with vertices, we have a queue with paths and the last vertex in the path represents the same vertex as in BFS.

Now based on the number of times this vertex appears in the path we can decide whether we should add it to the queue or not.
When a new vertex is added to queue it means that in a future while loop iteration, this vertex will be examined by the inner for loop and therefore, we will continue to search for more paths.

Scenario 1: Add to queue if the number of visits equals 0.
Scenario 2: Add to queue if the number of visits equals 0 or the vertex is a city.
Scenario 3: Add to queue if the number of visits equals 0 or the vertex is a city, or the village has a passport and its number of visits is less than 2.

(b)

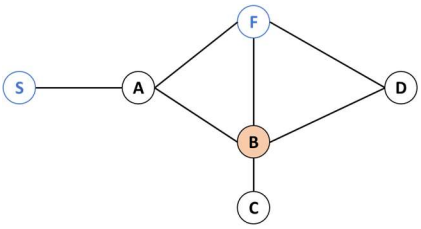
My proposed board configuration is the following.

```
Cities: [B]
Villages: [A, C, D]
Passport: []
S: [A]
F: [A, B, D]
A: [S, F, B]
B: [A, F, D, C]
C: [B]
D: [B, F]
```

My Python program will parse this text and extract the relevant information.
Insert cities, villages and villages with passports in simple arrays and create the board graph as an adjacency list.
The format shown above has to be respected in order for my program to work.

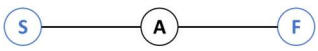
(d)

Test 1:



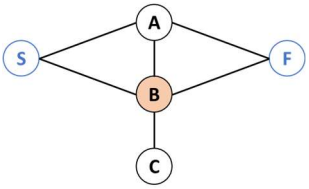
Same graph as the one provided in the assignment.
Includes one city and three villages.
From the output we can see how scenarios 1 and 2 produce different number of paths.
Scenario 2 includes all the paths from scenario 1.
In this test there is no village with passport so the output for scenario 2 and 3 is the same (as expected).

Test 2:



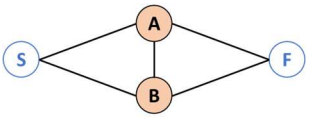
Simple board game with a single village connected to the start and finish.
The output is expected to be S -> A -> F

Test 3:



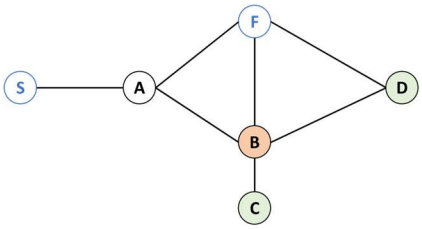
This game board includes two villages and one city. All the places are connected together (except start and finish).
When running scenario 2, we can expect to see paths that use 'B' more than once.
For example, S -> B -> A -> B -> C -> B -> F

Test 4:



This board game demonstrates a case with infinite paths. Since 'A' and 'B' are cities, we could travel back and forth between these two cities forever. S -> A -> B -> A -> B -> A -> …
The manufacturers of the game have to be careful with these types of boards. My algorithm can help them spot these infinite loops so that they can improve the game board and avoid infinite paths.

Test 5:



Same board as test 1 except that 'C' and 'D' are villages with passport.
We except paths that use 'C' and 'D' twice.
For example, S -> A -> B -> C -> B -> C -> B -> D -> B -> D -> F