Depth First Search (DFS):

Problem Statement:

Implementation of a function that takes in a problem parameter representing a graph and outputs a sequence of actions that can reach the goal state using depth first search.

Solution of the problem:

The generic steps used for solving the three problems are mentioned in short below:

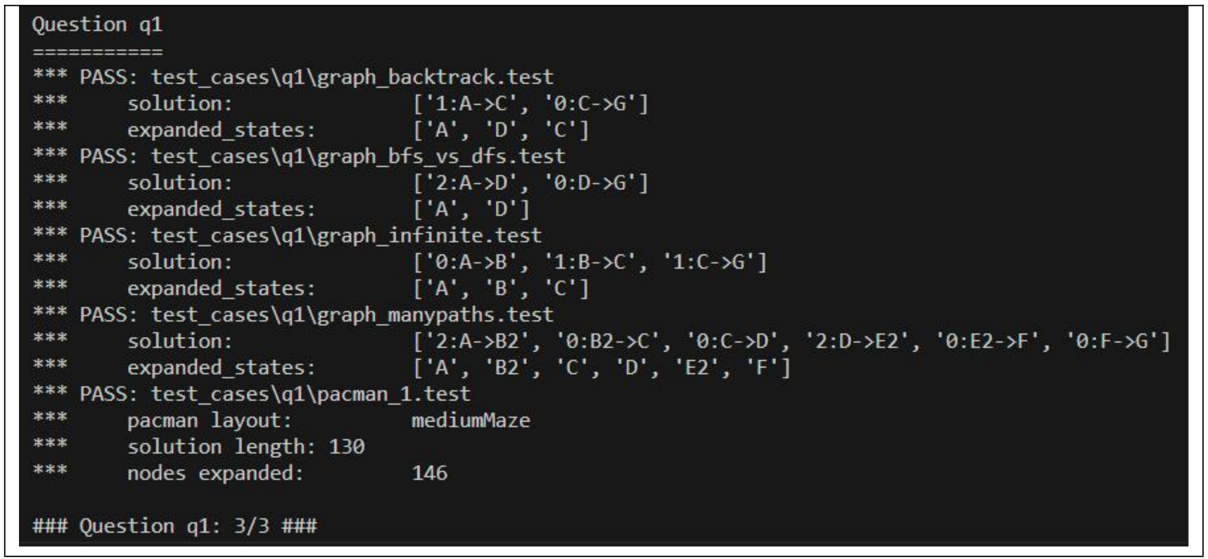
- A data structure termed as a "fringe" is used for storing the nodes.
- An array is used to keep track of already visited nodes whose successors have been obtained and pushed into the fringe.
- A loop is used to iterate the nodes in the fringe and add each of their successors into the fringe as well, given that they're not the goal state and the node has not been visited before.
- In each iteration, the current node is added to the array to ensure it is not visited again.
- When a goal state is found, the algorithm stops and returns the path followed to obtain the goal state. Otherwise it stops when the fringe is empty and returns None.

The fringe used in the DFS problem for storing the nodes is a stack. DFS explores the depth before continuing to the adjacent node and stack ensures this using its Last In First Out (LIFO) policy. The stack ensures that the latest pushed nodes are visited first i.e. the last successor, suppose X, of the node in the previous iteration is visited. The successor's successors are pushed into the stack again after performing goal test and whether it has been visited or not. In the next iteration, the last pushed successor of X is visited. This ensures the algorithm searches depth first.

If a successor which has already been visited, in case of nodes with multiple edges, the successor will already have been recorded in the visited array and the iteration will continue with the successor that was pushed before that. If all successors have been visited, the stack will now contain the node on the adjacent level with the node whose successors have been visited i.e. the previous depth level. Thus the only order being followed here is the order of the depth level of the nodes.

DFS does not consider the weight of the arcs of the graph. It also returns the first solution that it finds along a path, which may not be the most optimal path. There might be another optimal path which explores lower number of nodes to reach the solution which the DFS does not explore because it already returns the previous suboptimal solution it found.

Result:



```
Question q1
===========
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:               ['2:A->D', '0:D->G']
***     expanded_states:        ['A', 'D']
*** PASS: test_cases\q1\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_manypaths.test
***     solution:               ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 130
***     nodes expanded:         146

### Question q1: 3/3 ###
```

Breadth First Search (BFS):

Problem Statement:

Implementation of a function that solves the same problem as the previous task but using Breadth First Search Algorithm.

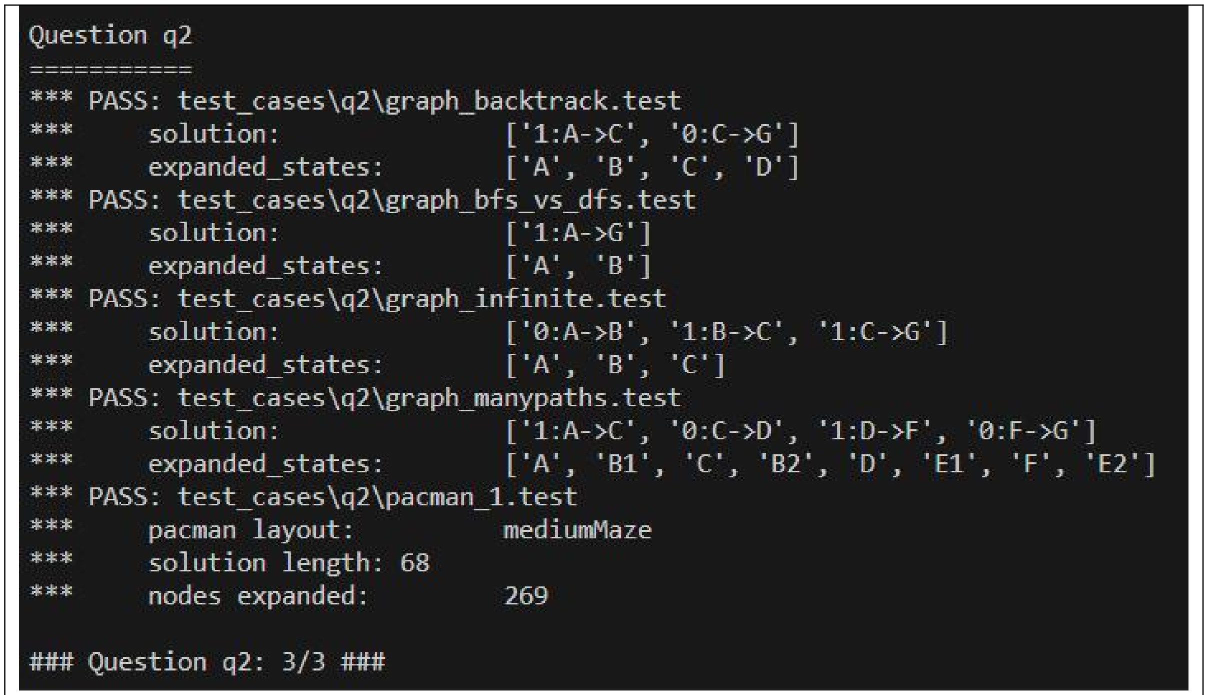
Solution of the problem:

This follows the steps of as the algorithm described in the first search problem, however it has some key differences due to difference in algorithm. BFS explores the adjacent nodes in the same depth level before exploring the next depth level. Thus it has to be ensured that the nodes in the fringe are visited in that particular order. As the data structure used to store the nodes partially determines in what order the node is accessed, a queue is used here.

The nodes are accessed in First In First Out (FIFO) policy due to the queue implementation. During the first iteration, the node that was first pushed is popped and its successors are pushed. In the next iteration, the first successors of the node in the previous iteration is popped and its successors are pushed. Since the nodes are accessed in the order they were pushed, the successors of the first node that was pushed i.e. the root node are all popped and their successors pushed before the next depth level is explored. This is what gives the algorithm its name “Breadth First Search” where it explores all nodes on the same depth level before moving onto their successors.

This search algorithm does not consider any weights in the arcs from one node to another since it is not considering the arc weights in the traversal of the nodes. BFS also finds the path with the shortest numbers of edges since it searches one depth level at a time for the solution and returns the solution at the lowest depth level (with the shortest path explored).

Result:



```
Question q2
===========
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 3/3 ###
```

Uniform Cost Search(UCS):

Problem Statement:

Implementation of uniform cost search algorithm that ensures that the path with the least total cost to the goal state.

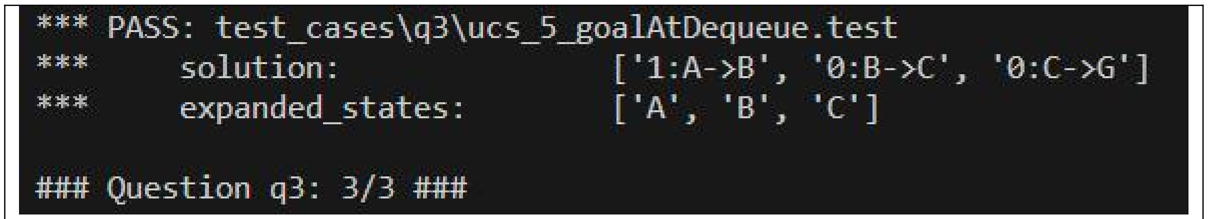
Solution of the problem:

In this search algorithm, the arc with the least cost possible is now selected at each iteration. It keeps track of the total cost and stores the nodes in such an order so that the arc with least cost is traversed next. The data structure that can do this is a priority queue, which can store the nodes according to increasing cost of the arcs and ensure the node with lowest cost is at the start of the queue.

In the first iteration, the successors of the root node is pushed into the queue and because of the workings of a priority queue, the successor which has the lowest cost arc is placed in the first of the queue. Following FIFO, in the next iteration, this successor is traversed next and all its successors are placed in the queue arranged in the ascending order of the arc cost. Each time in an iteration, the node with the lowest arc cost is picked until a goal state is reached where the cost required to reach the solution was the lowest. As observed, the depth levels of the nodes are not considered at all and only the cost is picked.

UCS takes a longer time to reach the solution since it expands in almost all directions to follow the arcs with the lowest cost. However this works for weighted graphs where not all the arcs have the same cost and the solution must be found with the least cost path. The number of nodes expanded for reaching a solution might be higher since UCS does not consider that to be a factor in its search.

Result:



```
*** PASS: test_cases\q3\ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 3/3 ###
```

Overall Challenges:

One of the main challenges was to comprehend the predefined functions and how to integrate them into the algorithm. Since DFS was implemented beforehand by the lab instructor, implementing BFS was fairly easy. The key was to understand which part of the DFS required modification so the algorithm could be changed to BFS. This became slightly trickier for the implementation of UCS since this required more modification and granular level of understanding of how the predefined functions actually worked.

However once I could relate my theoretical understanding with the implementation in python and approached it in steps, it became easier to understand what the predefined function did and what the expected outcome of the search algorithms were.