Course Number and Name: CSE 4618 Artificial Intelligence Lab	
Report: Lab 1 - Uninformed Search	
Student Name: Nafisa Maliyat	**Student ID:** 200042133
Date Of Submission: 31 January, 2024	**Submitted to:** Md. Bakhtiar Hasan, Assistant Professor, CSE

Overview:

This lab asked for implementation for search algorithms Breadth First Search, Depth First Search and Uniform Cost Search. There was three tasks provided with a usearch.zip file. After solving each task, Autograder was used to evaluate whether or not it was the correct solution.

On the next pages, I have mentioned the following :

- the problem statement
- solution of the problem
- the result of the problem
- Overall challenges of the lab

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 data structure stack is used for storing the nodes and an array is used to keep track of nodes that have visited already. DFS explores the depth before continuing to the adjacent node and stack ensures this using its Last In First Out (LIFO) policy, which will be elaborated later. The root node of the graph is pushed into the stack and a loop is started that continues until the stack is empty.

The node at the top of the stack (i.e. the root node at the first iteration) is popped and the current node and current path being followed is stored in variables. This state is checked to see if it is the goal state and if so, it is returned as the result. Next if the state has not already been visited, the successors of the state are obtained using a predefined function. This stores all possible states to go from the current state.

For each successor obtained, path being followed is stored along with the node within the stack. The stack ensures that the latest pushed nodes are visited first i.e. the nodes which are in the next depth level and have just been pushed into the stack will be explored first before the algorithm comes back to the previous depth level. After the successors have been stored in the stack, the current node of the loop is added to array of visited nodes to ensure the same node will not be visited again.

This process is repeated until the stack is empty or until the goal state has been reached. The function returns None for the first case and the path followed to reach the goal state for the second.

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

The rest of the steps are identical to the solution to the first problem statement for DFS. The nodes are accessed in First In First Out (FIFO) policy due to the queue implementation and for the current node, a check is performed for the goal state. If it is not a goal state, its successors are then obtained using the predefined function getSuccessors() and pushed into the queue. The current node is then saved to the array of visited nodes. This, again, continues until the goal state is reached or the queue is empty and similarly, returns the

path followed or None.

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:

The underlying process remains the same for this search algorithm as well. However the node with the least cost possible is now selected at each iteration, which requires keeping track of the total cost and storing the nodes in such an order so that the node with least cost is picked next. The data structure that can do this is a priority queue, which can store the nodes according to increasing cost and ensure the node with lowest cost is at the start of the queue.

Similar to the previous two problems, the root node is pushed into the priority queue and the nodes in the priority queue are iterated until either the queue is empty or the the goal state is reached. For each current node, the element - the node with the lowest cost - from the start of the queue is picked and set as current node. The check for goal state and whether it was already visited before is performed before obtaining its successors. The successors are then pushed into the priority queue and the current node is stored into the array of visited nodes.

As before, if the goal state is reached, the node is returned and if no such state is reached and the queue becomes empty before that, None is returned.

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.