## Introduction :

The tasks of this particular lab falls under a certain category of search problem which is Uninformed Search. Uninformed Search is the kind of search where the problem space is iterated by blindly following the algorithm steps without the prior information about the world state of the problem space.
3 types of Uninformed search is used in this lab to find a fixed food palette inside the maze. Those are :

1. Depth First Search (DFS)
2. Breadth First Search (BFS)
3. Uniform Cost Search (UCS)

We need to define the algorithms as solution in the ***search.py*** in their specific method and this ***search.py*** is used by the ***searchAgents.py*** which helps to simulate the agents which adopts the particularly defined algorithm to demonstrate a specific behaviour according to the algorithm while searching for the fixed food dot in the maze.
Also, each of the algorithms needs different types of data structures like Stack, Queue and Priority Queue as fringe. These data structures and their respective methods are defined in the ***utils.py***
Other necessary methods those are required for defining start state, check goal test, and get successors to expand the search tree are defined in the SearchProblem.
We needed to write the algorithms in the search.py in their specific position.

## Task - 1 ( Depth First Search ):

### Problem Analysis :

For the task -1, we need to find the fixed food dot in the maze by enabling pacman with the Depth First Search Capability. DFS uses Stack Data Structure as fringe which is a LIFO ( Last In First Out ) data structure. As this algorithm searches with respect to depth so it uses the stack to keep track of the next node to start the search when the algorithm reaches the dead end during the traversal ( Basically Backtracking purpose ).

**Solution Explanation :**

The solution of this task was demonstrated in the lab. In the demonstration, along with the fringe, a closed set “`closed()`” was used to keep track of the list of the states that is already visited by the algorithm so that those particular states don't get visited multiple times by the algorithm. But I used the “`closed []`” list as it was in the given solution in the Google Classroom. If this closed set or list isn’t maintained, it may lead to computation overhead or falling into a cycle in the search state graph- basically an infinite loop. The Stack data structure which is used as fringe is used from ***utils.py.*** The iterative process of DFS involves visiting states according to the plan. But it excludes the states those are are in the closed list. DFS effectively explores the leftmost unvisitied branches by iterating over the successors of each state within the fringe. It doesn’t always return the optimal path or plan because it executes the search by going deep using the leftmost branch

## Task - 2 ( Breadth First Search ) :

**Problem Analysis :**

For the task -2, we need to find the fixed food dot in the maze by enabling pacman with the Breadth First Search Capability. In BFS, we need the Queue data structure as fringe. BFS explores the maze traversing all the neighbouring nodes of a node before traversing to the next level.. BFS finds the optimal path when all the costs of going to a node from another node is same. As in this fixed food dot problem, all costs are equal so BFS return the most optimal path/plan.

**Solution Explanation :**

The solution is pretty similar to the DFS one. We just introduced Queue instead of the Stack as fringe. The Queue is used there because it maintains the order in which nodes are getting discovered and traversed. It ensures the orderly processing of the all nodes in the maze meaning nodes that were discovered first will be traversed first. So we traversed the nodes level by level in BFS unlike going to the deepest level of a branch in DFS. The other algorithmic steps are similar to the DFS problem.

## Task - 2 ( Uniform Cost Search ) :

### Problem Analysis :

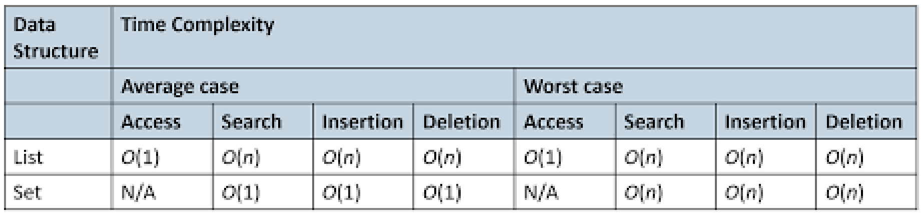
For the task -3, we need to find the fixed food dot in the maze by enabling pacman with the Uniform Cost Search Capability. In UCS, a Priority Queue Data Structure is needed as fringe. UCS considers the cost of reaching each node unlike DFS or BFS. Because of this UCS returns the most optimal path by giving priority to nodes which have the lower accumulated cost while traversing through the space state graph.

### Solution Explanation :

The solution is similar to DFS and BFS to some extent but here we needed to associate accumulated cost with each node while initialzing the nodes and traversing the nodes because in UCS we are expanding the search tree with respect to the minimum accumulated cost of going to any particular node. As per the definition in the util.py, the priority queue used in the solution prioririzes the minimum cost first, so our start state is given 0 as the accumulated cost which gives it the highest priority to start with. In DFS or BFS we were considering State and Plan only but in UCS we are considering Cumulative Cost along with them while determining the next node to traverse. Other properties of the algorithm remains similar to the DFS and BFS.
UCS will return the most optimal (lowest cost) path from the start position of the pacman to the position of the fixed food dot in the maze.

## Challenges Faced and Area of Improvement:

As the DFS was demonstrated line by line in the lab and the properties of BFS, UCS was told beforehand, I didn't face any particular problem in this lab.
But I used list in my solution instead of Set (which was preferred in the lab) to keep track of the visited nodes. Later I realized using Set was the most optimal way to approach all the problems as we needed to search for an element's presence in the closed list.



| Data Structure | Time Complexity | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Average case | | | | Worst case | | | |
| | Access | Search | Insertion | Deletion | Access | Search | Insertion | Deletion |
| List | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ | $O(1)$ | $O(n)$ | $O(n)$ | $O(n)$ |
| Set | N/A | $O(1)$ | $O(1)$ | $O(1)$ | N/A | $O(n)$ | $O(n)$ | $O(n)$ |

Here in the attached table, it is clearly seen that Set incurs less cost when we search for an element if that is present or not **( O(1) < O(n) ).**