Department of Computer Science and Engineering

Islamic University of Technology (IUT)

A subsidiary organ of OIC

Lab Report: Uninformed Search

CSE 4618: Artificial Intelligence

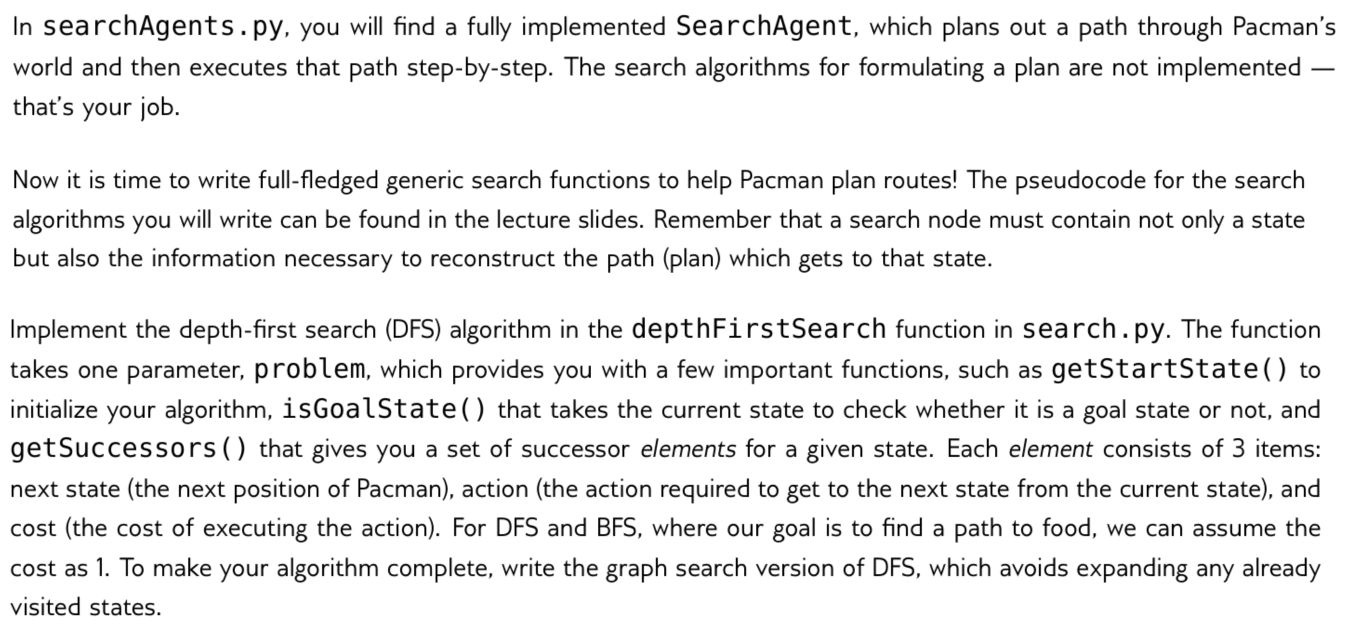
Name	**: Mashrur Ahsan**
Student ID	**: 200042115**
Section	**: 1 (SWE)**
Semester	**: Winter (6th)**
Academic Year	**: 2023-24**
Date of Submission	**: 01/02/2024**
Lab No.	**: 1**

Introduction:

This lab is an introduction to the concept of uninformed search. We would need to build general search algorithms and apply them to Pacman scenarios.

Problem Statements:

1. Finding a Fixed Food Dot using Depth First Search Algorithm



In `searchAgents.py`, you will find a fully implemented `SearchAgent`, which plans out a path through Pacman's world and then executes that path step-by-step. The search algorithms for formulating a plan are not implemented — that's your job.

Now it is time to write full-fledged generic search functions to help Pacman plan routes! The pseudocode for the search algorithms you will write can be found in the lecture slides. Remember that a search node must contain not only a state but also the information necessary to reconstruct the path (plan) which gets to that state.

Implement the depth-first search (DFS) algorithm in the `depthFirstSearch` function in `search.py`. The function takes one parameter, `problem`, which provides you with a few important functions, such as `getStartState()` to initialize your algorithm, `isGoalState()` that takes the current state to check whether it is a goal state or not, and `getSuccessors()` that gives you a set of successor *elements* for a given state. Each *element* consists of 3 items: next state (the next position of Pacman), action (the action required to get to the next state from the current state), and cost (the cost of executing the action). For DFS and BFS, where our goal is to find a path to food, we can assume the cost as 1. To make your algorithm complete, write the graph search version of DFS, which avoids expanding any already visited states.

The task is to implement the depth-first search (DFS) algorithm for the Pacman game

Analysis of the Problem Statement:

The fully implemented Search Agent has the ability to plan a path through Pacman's world but lacks the implementation logics of the search algorithms. The task is to implement the DFS algorithm in the Depth First Search function.

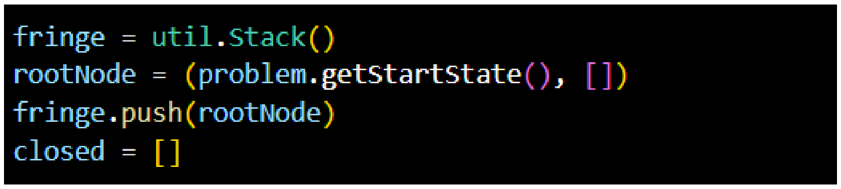
There are useful data structures for implementing search algorithms for our disposal. There is a helper class that provides important functions that helps us get the start state, get the successors and check whether a node is a goal node or not.

The problem statement also asks us to consider the case when a node might already be visited once. We have to implement the algorithm in such a way that it avoids expanding already visited nodes - ensuring that the algorithm doesn't get stuck in infinite loops.

The DFS algorithm needs to return a sequence of legal moves to the goal. To test the correctness of the solution all the test cases need to be passed.

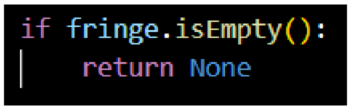
<u>The Solution & Explanation:</u>

Here, we are using the stack data structure as our fringe to implement this DFS algorithm. Stack allows the algorithm to explore as deep as possible before backtracking.

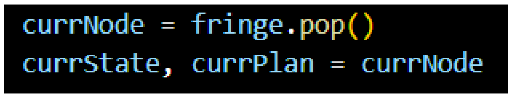


```
fringe = util.Stack()
rootNode = (problem.getStartState(), [])
fringe.push(rootNode)
closed = []
```

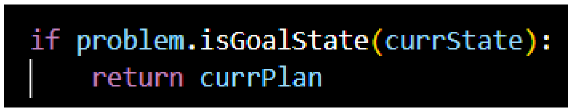
The root node is initialized as a tuple of the start state and a plan (sequence of legal actions). In this case, since it's the start node, the plan list is empty. This root node represents the current node in the search. There's also a list that keeps track if the nodes that's already been visited (once pulled out of the fringe). The closed list is keeping track of that.



```
if fringe.isEmpty():
    return None
```

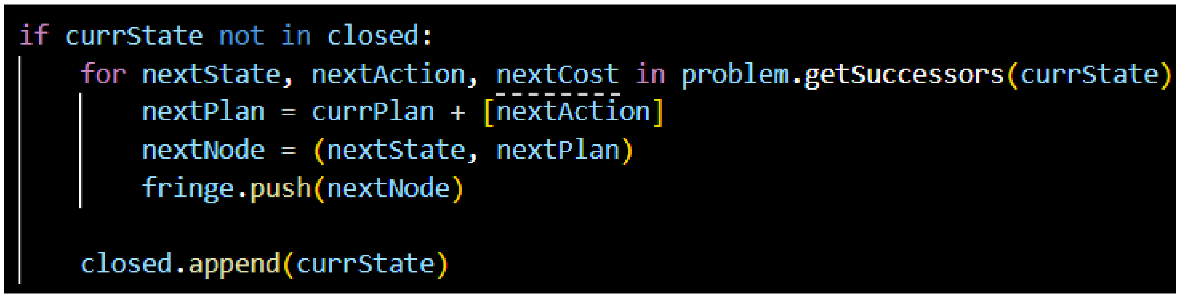


```
currNode = fringe.pop()
currState, currPlan = currNode
```



```
if problem.isGoalState(currState):
    return currPlan
```

The main loop exits when the fringe is empty or the goal node is reached. If the goal node is reached then the it will return the current plan. Another case that we have to consider is that, whether the current node is already visited or not. Only expand if it was not visited before. These are the cases we have to consider here.



```
if currState not in closed:
    for nextState, nextAction, nextCost in problem.getSuccessors(currState)
        nextPlan = currPlan + [nextAction]
        nextNode = (nextState, nextPlan)
        fringe.push(nextNode)

    closed.append(currState)
```

If a non-goal node wasn't visited before then we look for the next node and check if it's the goal node or not. For that we have to expand. Now, how do we expand? We expand by getting the successor nodes of the current non-goal node. We expand the current plan and move on to the next node by pushing it inside the fringe. Moreover, we need to append the current non-goal node the closed list since it's now visited.

Now, this process repeats until we find the goal node or the fringe exhausts. This is how the DFS algorithm works here.

Challenges:

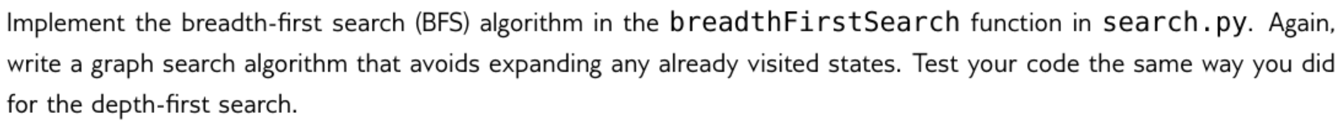
Here we are using lists, tuples, list of tuples, lists inside tuples etc. Sometimes I had trouble figuring out which operation is dealing with or returning which data structure and how to handle each data structure in a proper way.

Findings:

How DFS worked in the Pacman's world. What the concept of uninformed search worked in Pacman's world. Implementation logic behind uninformed search in the Pacman's world – DFS.

How lists, tuples, list of tuples, lists inside tuples etc. worked together to formulate a search algorithm. Also came to know the various cases that one has to consider while implementing a search algorithm.

2. Finding a Fixed Food Dot using Breadth First Search Algorithm


Implement the breadth-first search (BFS) algorithm in the `breadthFirstSearch` function in `search.py`. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for the depth-first search.

The task is to implement the breadth-first search (BFS) algorithm for the Pacman game

Analysis of the Problem Statement:

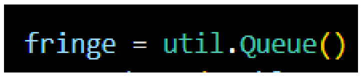
The fully implemented Search Agent has the ability to plan a path through Pacman's world but lacks the implementation logics of the search algorithms. The task is to implement the BFS algorithm in the Breadth First Search function.

Other portions of the problem statement are similar to the first algorithm that we had to implement. Instead of DFS, now we are performing BFS.

Similar to the DFS algorithm, BFS needs to return a sequence of legal moves to the goal. To test the correctness of the solution all the test cases need to be passed.

The Solution & Explanation:

Here, we are using the Queue data structure as our fringe to implement the BFS algorithm. a Queue is used to explore nodes in a breadth ward motion. BFS systematically visits all the neighbors of a node before moving on to the next level of nodes. The queue follows the First-In-First-Out (FIFO) principle, meaning that the node that has been in the queue the longest is the first one to be processed.


```
fringe = util.Queue()
```

The root node initialization, tuple formation, how the fringe and the closed list would work is similar to the DFS algorithm.

Again, the main loop, how the visited nodes are handled, how the nodes are expanded, what conditions we have to check every time is also similar to the DFS algorithm. This is how the BFS algorithm works here.

Challenges:

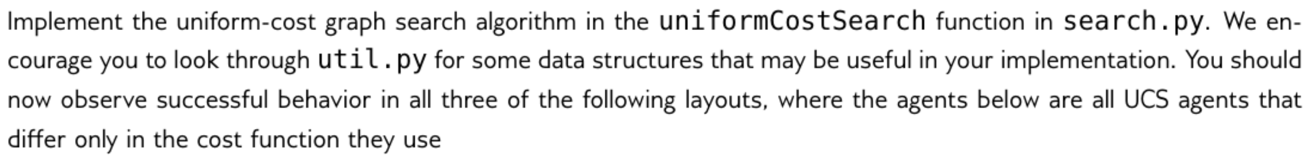
No significant challenges faced since I was able to overcome the challenges mentioned in the first task so that a proper formulation of the solution would be possible.

Findings:

How BFS worked in the Pacman’s world. What the concept of uninformed search worked in Pacman’s world. Implementation logic behind uninformed search in the Pacman’s world – BFS.

How lists, tuples, list of tuples, lists inside tuples etc. worked together to formulate a search algorithm. Also came to know the various cases that one has to consider while implementing a search algorithm.

3. **Finding a Fixed Food Dot using Uniform Cost Search Algorithm**


Implement the uniform-cost graph search algorithm in the `uniformCostSearch` function in `search.py`. We encourage you to look through `util.py` for some data structures that may be useful in your implementation. You should now observe successful behavior in all three of the following layouts, where the agents below are all UCS agents that differ only in the cost function they use

The task is to implement the uniform cost search (UCS) algorithm for the Pacman game

Analysis of the Problem Statement:

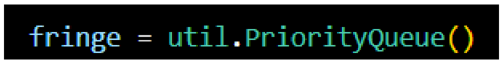
The fully implemented Search Agent has the ability to plan a path through Pacman's world but lacks the implementation logics of the search algorithms. The task is to implement the UCS algorithm in the Uniform Cost Search function.

Other portions of the problem statement are similar to the first or the second algorithm that we had to implement. Instead of DFS and BFS, now we are performing UCS.

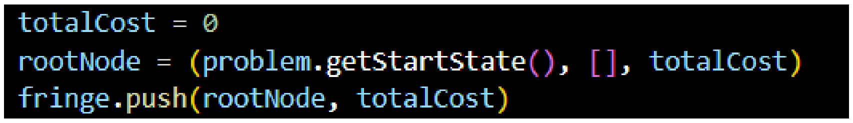
Similar to the DFS or BFS algorithms, UCS needs to return a sequence of legal moves to the goal. This represents the sequence of actions or moves taken by the agent to reach the goal. This is the cumulative cost associated with the path from the start state to the goal state. The UCS algorithm aims to minimize this cost. To test the correctness of the solution all the test cases need to be passed.

The Solution & Explanation:

Here, we are using the Priority Queue data structure as our fringe to implement this UCS algorithm. Priority Queue allows us to the explore nodes based on their path cost. The priority queue follows the priority-first principle, where nodes with lower costs have higher priority and are processed before nodes with higher costs.


```
fringe = util.PriorityQueue()
```

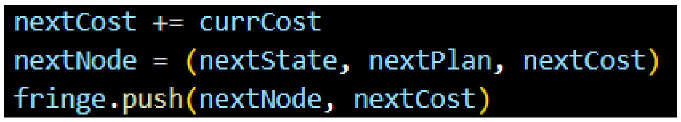
The root node initialization, tuple formation, how the fringe and the closed list would work is similar to the DFS or BFS algorithm. Except, now, we are also keeping track of the total cost.



```
totalCost = 0
rootNode = (problem.getStartState(), [], totalCost)
fringe.push(rootNode, totalCost)
```

So, there is another attribute inside the tuple. When we push the total cost, that denotes the cost it takes to reach that node from the start node.

Again, how the main loop works, how the visited nodes are handled, how the nodes are expanded, what conditions we have to check every time is also similar to the DFS or the BFS algorithm.



```
nextCost += currCost
nextNode = (nextState, nextPlan, nextCost)
fringe.push(nextNode, nextCost)
```

The only thing that we are doing more is handling the cost attribute. We are calculating the cost every time we reach a non-goal node. This is how the UCS algorithm works here.

Challenges:

No significant challenges faced since I am already acquainted on how the DFS and the BFS works.

Findings:

How UCS worked in the Pacman's world. What the concept of uninformed search worked in Pacman's world. Implementation logic behind uninformed search in the Pacman's world – UCS.

How lists, tuples, list of tuples, lists inside tuples etc. worked together to formulate a search algorithm. Also came to know the various cases that one has to consider while implementing a search algorithm.