Department of Computer Science and Engineering

Islamic University of Technology (IUT)

A subsidiary organ of OIC

CSE 4618: Artificial Intelligence

Lab Report 01

Name	**:**	**M M Nazmul Hossain**
Student ID	**:**	**200042118**
Semester	**:**	**6th**
Academic Year	**:**	**2022-2023**
Date of Submission	**:**	**12.02.2024**

Problem-1 (Depth First Search)

Analysis

The first problem was to define a search function that implemented the DFS algorithm to act as a search agent for the Pacman problem. It provided a parameter, the problem, which gave us helpful functions like getStartState to start formulating the search function, the getSuccessor function to get the subsequent possible states based on all the possible actions for Pacman, and the isGoalState function that checks whether the current state Pacman is in is the goal state or not.

Solution

The problem was solved by Bakhtiar sir as a demonstration.

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print("Start:", problem.getStartState())
    print("Is the start a goal?", problem.isGoalState(problem.getStartState()))
    print("Start's successors:", problem.getSuccessors(problem.getStartState()))
    """
    "*** YOUR CODE HERE ***"
    # util.raiseNotDefined()
    fringe = util.Stack()
    rootNode = (problem.getStartState(), [])
    fringe.push(rootNode)
    closed = []

    while True:
        if fringe.isEmpty():
            return None

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

        if problem.isGoalState(currState):
            return currPlan

        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)
```

Explanation

The first problem was solved by Sir. Since it required the DFS algorithm, the solution began by declaring the fringe as a **stack**. The root node was initiated using the problem.getStateState function and an array which stored the plan of actions taken to reach that node (which was kept empty since it is the root node). The root node is pushed inside the fringe.

The graph search approach was also followed, seeing that sir kept a closed set, to track the nodes that had been previously expanded upon in order to not waste resources. The root node is pushed into the fringe and then a loop starts.

The loop checks first whether the fringe is empty or not, then it pops the first node in the fringe to extract the current state and the path or plan followed to get to that state. Then it checks whether the current state was the Goal State or not. If it is the goal state, it returns the current path, or plan. Otherwise, it uses the getSuccessor function to get all the subsequent states and pushes them into the fringe one by one, while adding the action taken to get to that state to the path or plan and finally add the state inside the closed set if the current state was not in the closed set already.

The property of the fringe being made of a stack ensures that the last node in is the first one popped. So, the fringe pops out the most recent node pushed in. This means when a node pushes in its successor nodes, the next node popped when the loop comes back is one of the successor nodes. Therefore, the search node explores the graph depth first.

In this implementation, the cost of the nodes didn't come into play while traversing the graph, because we were traversing the nodes based on the levels. So, the cost of each node could be 1, 100, same or different, the resulting path would remain the same. It should be noted that this approach wouldn't return the optimal path in all situations. Since this is the graph search approach, DFS is complete provided enough time and that a solution exists.

Challenges

No challenges were faced while doing this as it was solved by Sir.

Problem-2 (Breadth First Search)

Analysis

The second problem was to define a search function that implemented the BFS algorithm which will act as a search agent for the pacman problem. It provided a parameter, the problem which gave us helpful functions like getStartState to start formulating the search function, the getSuccessor function to get the subsequent next Possible states based on all the possible actions for pacman, and the isGoalState function which checks whether the current state pacman is in is the goal state or not.

Solution

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    # util.raiseNotDefined()
    fringe = util.Queue()
    rootNode = (problem.getStartState(), [])
    fringe.push(rootNode)
    closed = []

    while True:
        if fringe.isEmpty():
            return None

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

        if problem.isGoalState(currState):
            return currPlan

        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)
```

Explanation

The second problem required the BFS algorithm, the solution began by declaring the fringe as a **queue**. The root node was initiated using the problem.getStateState function and an array which stored the plan of actions taken to reach that node (which was kept empty since it is the root node). The root node is pushed inside the fringe.

The graph search approach was followed here as well to track the nodes that had been previously expanded upon in order to not waste resources expanding the same nodes again. The root node is pushed into the fringe and then a loop starts.

The loop checks first whether the fringe is empty or not, then it pops the first node in the fringe to extract the current state and the path or plan followed to get to that state. Then it checks whether the current state was the Goal State or not. If it is the goal state, it returns the current path, or plan. Otherwise, it uses the getSuccessor function to get all the subsequent states and pushes them into the fringe one by one, while adding the action taken to get to that state to the path or plan and finally add the state inside the closed set if the current state was not in the closed set already.

The property of the fringe being made of a queue ensures that when a node is popped from the fringe, it will be in the sequential order in which it was pushed in. So, the first node in is the first node out. This means, that when a node pushes in its successors at level A, and the loop expands one of those successor nodes, and pushes in its successors at level A+1, the fringe will first expand and push in the successor nodes of all the nodes at level A, before starting to pop nodes found at level A+1. So, the graph is traversed based on levels, breadth first.

In this implementation, the cost of the nodes didn't come into play while traversing the graph, because we were traversing the nodes based on the levels. So, the cost of each node could be 1, 100, same or different, the resulting path would remain the same. It should be noted that BFS would return the optimal solution if the cost of all the nodes were uniform, and it is also complete. However, this approach is expensive since it requires a lot of memory and isn't suitable in situations where the branching factor is too large.

Challenges

No challenges were faced while doing this as most of the code was already implemented by sir in the DFS implementation. I simply changed the fringe from a Stack to a Queue.

Problem-3 (Uniform Cost Search)

Analysis

The third problem was to define a search function that implemented the UCS algorithm which will act as a search agent for the pacman problem. It provided a parameter, the problem which gave us helpful functions like getStartState to start formulating the search function, the getSuccessor function to get the subsequent next Possible states based on all the possible actions for pacman, and the isGoalState function which checks whether the current state pacman is in is the goal state or not.

Solution

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    # util.raiseNotDefined()
    fringe = util.PriorityQueue()
    rootNode = (problem.getStartState(), [], 0)
    fringe.push(rootNode, 0)
    closed = []

    while True:
        if fringe.isEmpty():
            return None

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

        if problem.isGoalState(currState):
            return currPlan

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

            closed.append(currState)
```

Explanation

The third problem required the UCS algorithm, the solution began by declaring the fringe as a **PriorityQueue**. Specifically, a minimum priority queue. The root node was initiated using the problem.getStateState function and an array which stored the plan of actions taken to reach

that node (which was kept empty since it is the root node) and the cumulative cost required to reach the root node which is logically, 0. The root node is pushed inside the fringe alongside the cost required to reach the root node.

The graph search approach was followed here as well to track the nodes that had been previously expanded upon in order to not waste resources expanding the same nodes again. The root node is pushed into the fringe and then a loop starts.

The loop checks first whether the fringe is empty or not, then it pops the first node in the fringe to extract the current state, the path or plan followed to get to that state and cumulative cost required to reach that state (backward cost). Then it checks whether the current state was the Goal State or not. If it is the goal state, it returns the current path, or plan. Otherwise, it uses the getSuccessor function to get all the subsequent states and pushes them into the fringe one by one, while adding the action taken to get to that state to the path or plan alongside the updated cost to reach each successor and finally add the state inside the closed set if the current state was not in the closed set already.

The property of the fringe being made of a minimum priority ensures that when a node is popped from the fringe, it will pop the node with the lowest backward cost first, so that the optimal path can be found where the nodes have varying costs. An interesting outlook could be, when initializing the problem, the cost at the root node could be given whatever we want to, as the subsequent nodes will have a backward cost according to the root node (1996+5, 1996+5+7 etc) , as it will always be popped since at the beginning it is the only node in the fringe. However, for logical clarity, it is more sensible to use 0. It should be noted that UCS would return the optimal solution in all cases, and it will be complete, given a solution exists.

Challenges

No challenges were faced while doing this as most of the code was already implemented by sir in the DFS implementation. I simply changed the fringe from a Stack to the already provided Priority Queue, and considered the backward cost in the implementation.