CSE 4618: Artificial Intelligence Lab

Lab 1

Name: K.M.Tahlil Mahfuz Faruk

ID:200042158

Islamic University of Technology, OIC

Gazipur, Bangladesh

Feb 13, 2024

Question 1 (Finding a Fixed Food Dot using Depth First Search):

Introduction

The problem asks us to implement the Depth First Search algorithm to find the food dot.

Analysis of the problem

- In the search.py file there is a function called depthFirstSearch
- We need to complete the depthFirstSearch function that gives us the solution to reach the food dot exploring the deepest nodes first strategy.
- As we are exploring the deepest node first and it always explores the leftmost leaf node, it can be said that it might be an optimal solution if the goal is situated in the leftmost portion of the tree that is being traversed first.
- On the other hand, if the goal is at the upper layer but it is on the narrowest portion of the tree then DFS would be very inefficient as it needs to explore all the leftmost nodes to the leaf first then come to the other side where the goal is situated. In this case, DFS will not be the optimal solution.
- Consequently, DFS might be or might not be the optimal solution. It depends on the goal that we are trying to achieve.
- Also we have seen after running pacman.py that there is a cost for movement or action of pacman. In this case, It is likely that DFS won't be the optimal solution as it's not considering the cost of action.

Explanation of the solutions

- In depthFirstSearch function, we used a stack as a fringe. Stack data structure was implemented in the util.py file.
- Using the getStartState() we will get the starting node and we will start our search from that node.
- Each time we will check if we have reached the goal or not.
- We kept a closed set that contains the visited nodes, because we don't want to visit the already explored nodes again. That will increase the complexity of the algorithm.
- If the current state is not visited then we update the fringe and continue exploring using the getSuccessors function.
- Iteratively this strategy will get us to the goal.

Challenges and Solutions

During this problem no significant problem was encountered.

Interesting Findings

In the DFS code that we were provided, to keep track of the visited nodes there was an array. But it will keep duplicates of the visited nodes. It will not be space efficient. But during the lab demonstration our course teacher used a set as it will not keep duplicates and space complexity will be reduced. We have also used a set rather than an array.

Question 2 (Breadth First Search):

Introduction

The problem asks us to implement the Breadth First Search algorithm to find the food dot.

Analysis of the problem

- In the search.py file there is a function called breadthFirstSearch
- We need to complete the function that gives us the solution to reach the food dot exploring the narrowest nodes first strategy.
- As we are exploring the narrowest nodes first we will explore the upper nodes first then get to the lower nodes.
- This might be a good solution if our goal is at any of the upper layers. The algorithm might be optimal.
- On the other hand, if the goal is at any of the deeper nodes then it won't be an optimal solution.
- Consequently, like DFS, BFS might or might not be the optimal solution. It depends on the problem that we are trying to solve.
- For the pacman.py program, considering the cost of an action, BFS won't be the optimal solution as it doesn't consider the cost after performing an action in the algorithm.

Explanation of the solutions

- In the depthFirstSearch function, we used a queue as a fringe. Queue data structure was implemented in the util.py file.
- Using the getStartState() we will get the starting node and we will start our search from that node.
- Each time we will check if we have reached the goal or not.

- We kept a closed set that contains the visited nodes, because we don't want to visit the already explored nodes again as that will increase the complexity of the algorithm.
- If the current state is not visited then we update the fringe and continue exploring using the getSuccessors function.
- Iteratively this strategy will get us to the goal.
- It's quite similar to DFS. Only the fringe is changed to a queue.

Challenges and Solutions

During this problem no significant problem was encountered.

Interesting Findings

Similarly as DFS we used a set rather than an array to keep track of the visited nodes. That decreases the space complexity.

Question 3 (Varying the Cost Function):

Introduction

The problem asks us to implement UCS(Uniform Cost Search) that will consider the cost of action by the pacman and find the best path to get to the food dot.

Analysis of the problem

- In the search.py file there is a function called uniformCostSearch
- We need to complete the function that gives us the solution to reach the food dot considering each action of pacman has some cost.
- In this solution the idea is to get to the current state to the next state that has the lowest cost.
- This should return the optimal solution.
- But the solution might not be the best solution. Because we are tracking the visited nodes using a closed set. There might be a case where an explored node might get us to a more cost efficient solution than that we can currently find.
- But compared to BFS and DFS, UCS will provide a more optimal solution.

Explanation of the solutions

- In the uniformCostSearch function, we used a PriorityQueue as a fringe. PriorityQueue data structure was implemented in the util.py file.

- Using the getStartState() we will get the starting node and we will start our search from that node.
- Each time we will check if we have reached the goal or not.
- We kept a closed set that contains the visited nodes, because we don’t want to visit the already explored nodes again. Because that will increase the complexity of the algorithm.
- If the current state is not visited then we need to extract the nextCost and update the fringe and continue exploring using the getSuccessors function.
- If we explore the priorityQueue implementation in util.py, we will find that it keeps the lowest priority/cost item at the top. That is retrieved first while popping the priorityQueue. The popped element becomes our next state.
- Iteratively this strategy will get us to the goal.

Challenges and Solution

- While solving the problem I didn't check the implementation of the PriorityQueue in the util.py and I didn't update the parameters of priorityQueue and the fringe according to that.
- **Solution:** After checking the implementation I updated the parameters and the fringe accordingly. That solved the issue.

Interesting Findings

As interesting finding we can mention the understanding though we are considering the lowest cost of the next state, in some cases we might find that another path/solution is more efficient as we are considering a visited node set to track the visited nodes.