Islamic University of Technology

Department of Computer Science and Engineering

Artificial Intelligence Lab

Course Code- CSE 4618

Prepared By:
Nazia Krim Khan Oishee
ID: 200042137

Date: February 1, 2024

Contents

1 Introduction

The objective of this lab was to construct general search algorithms and apply them to various Pacman scenarios.

- Pacman resides in a vibrant maze filled with winding corridors and delicious round treats. Efficient navigation through this world marks the beginning of Pacman's quest to master his surroundings..
- Our Pacman agent navigates through a maze environment to achieve certain goals, such as reaching a particular location and efficiently collecting food.
- We were given a zip file containing all the required supporting files. Among all the files, we only needed to edit search.py file, where all of our search algorithms would reside.
- There were other relevant files such as searchAgents.py, pacman.py, game.py, util.py. Among these files util.py included a number of helpful data structures for implementing search methods.
- Similar to Lab 0, we used Autograder for assessment.

2 Problem Analysis

In the lab, we were assigned to do three tasks.

2.1 Question1: Depth First Search

First question required us to find a fixed food dot using DFS. There existed a completely implemented SearchAgent in the searchAgents.py file. The agent develops a strategy to traverse through Pacman's environment and execute that path gradually. However,the path constructing search algorithm was not implemented . Implementing the algorithm was our first task.

There were several key instructions suggested for this assignment:

- A search node must include a state and the details required to build the path leading to that state.
- Additionally, all our search functions must provide a list of actions that would guide the agent from the start to the goal. These actions must consist of valid moves. Here valid moves indicate logical directions without traversing through the walls.
- Also, our function must utilize the Stack, Queue, and PriorityQueue data structures available in util.py file.

We were to implement the depth-first search algorithm in the depthFirstSearch function in search.py. The function had parameter named problem. It provided several important functions including getSuccessors(), isGoalState() and getStartState().

2.2 Question2: Breadth First Search

Our second task was to implement the BFS algorithm.

2.3 Question3: Varying the Cost Function

BFS finds a fewest-actions path to the goal, but we want to find paths that has the minimum costs.

So, our third task was to implement the UCS algorithm . There were some data structures in util.py that were useful for our implementation.

3 Solution Explanation

3.1 Question1

The function depthFirstSearch performs in the following way:

- A stack named fringe is initialized to store nodes for exploration.
- A root node is created with the start state obtained from the problem's getStartState() method and an empty plan. This root node is pushed onto the stack.
- To maintain track of the nodes visited and avoid revisiting the nodes already visited, a list is initialized.
- Afterwards, the function goes into an infinite loop that keeps exploring nodes until it finds a solution or the stack becomes empty.
- The function returns None if there is no goal state and the stack is empty.
- The state and plan of a node are extracted by popping out of the stack.
- The function returns the plan that lead to the goal if the current state is the target state.
- The function explores the successor states provided the current state is not the target state and the present state hasn't been explored yet.
- For each successor of the current state obtained through the getSuccessors() method:
  - A new plan is created by adding the current plan with the action from the successor.
  - A new node is formed with the successor's state and the new plan.
  - The new node is pushed onto the stack for further exploration.
- The current state is marked as visited by adding it to the closed list to avoid revisiting.

The function repeats this process until a goal state is found or the stack becomes empty. The use of a stack ensures that the function explores paths depth-wise. The closed list prevents revisiting states to optimize efficiency.

3.2 Question2

The function breadthFirstSearch performs in the following way:

- A queue (fringe) is initialized to store nodes for exploration.
- A root node is created with the start state obtained from the problem and an empty plan. This root node is enqueued onto the queue.
- A list (closed) is initialized to track visited states.

- Until a solution is found or the queue is empty, the function goes into an endless loop and keeps exploring nodes.
- The algorithm yields None if there is no target state and the queue is empty.
- A node is taken removed of the queue and its plan and state are taken out for inspection.
- The function returns the goal's path plan if the current state is a goal state.
- The function moves on to look into successors provided the current state hasn't been visited previously and it is not a goal state.
- For each successor of the current state obtained from the problem:
  - A new plan is created by extending the current plan with the action from the successor.
  - A new node is formed with the successor's state and the extended plan.
  - The new node is enqueued onto the queue for further exploration. Visited State Tracking:
- The current state is marked as visited by adding it to the closed list to avoid revisiting.

The function repeats this process until a goal state is found or the queue becomes empty. The use of a queue ensures that the algorithm explores paths level by level.

3.3 Question3

The function uniformCostSearch works the following way:

- A priority queue (fringe) is initialized to store nodes for exploration. Nodes are prioritized based on the total cost from the start state to the current state.
- A root node is created with the start state obtained from the problem and an empty plan. This root node is enqueued onto the priority queue with a priority of 0 (initial cost).
- A list (closed) is initialized to track visited states.
- The function enters an infinite loop to continuously explore nodes until a solution is found or the priority queue is empty.
- If the priority queue is empty (fringe.isEmpty()), and no goal state is found, the function returns None.
- After that a node is removed from the priority queue, its plan and state are taken out for inspection.
- The function returns the path leading to the goal, if the current state is a goal state.
- The function moves on to explore the successors, if the current state has never been visited before.
- For each successor of the current state obtained from the problem:
  - A new plan is created by extending the current plan with the action from the successor.
  - A new node is formed with the successor's state and the extended plan.
  - The total cost of the new plan is calculated using the problem.getCostOfActions() function.
  - The new node is enqueued onto the priority queue with its total cost as the priority for further exploration.

- The current state is marked as visited by adding it to the closed list to avoid revisiting.

The use of a priority queue ensures that nodes with lower total costs are explored first. The function repeats this process until a goal state is found or the priority queue becomes empty.

4 Findings and Insights

4.1 Question 1

Findings:

- **Order of Exploratio:** DFS explores paths depth-wise, exploring nodes at the deepest level before moving on to shallower levels.
- **Completeness:** DFS may not find a solution if the search space is infinite or if there is a loop in the graph.
- **Memory Usage:** The memory requirements for DFS can be significant for deep search spaces, as it needs to keep track of the entire path from the root to the current node.
- **Path Optimality:** DFS does not guarantee the shortest path. It might find a solution, but it may not be an optimal one.

Insights:

- **Trade-off Between Memory and Completeness:** DFS offers completeness for finite search spaces but might not be suitable for infinite spaces due to memory constraints.
- **Suboptimal Solution:** DFS may not always find the shortest path to the goal.
- **Applicable Scenarios:** DFS performs well in scenarios with deep solutions.

4.2 Question2

Findings:

- **Order of Exploratio:** BFS explores paths level by level. It prioritizes nodes at shallower levels before moving on to deeper levels.
- **Completeness:** BFS is complete for finite search spaces and guarantees finding the shallowest solution.
- **Memory Usage:** Memory requirements for BFS can be significant, especially for wide search spaces.
- **Path Optimality:** BFS ensures that the shortest path, is found to the destination in terms of path length.

Insights:

- **Trade-off Between Memory and Completeness:** BFS also involves a trade-off between memory usage and completeness.
- **Optimal Solution:** Being able to find the shortest path makes BFS an optimal solution in terms of length of the path.
- **Applicable Scenarios:** BFS might be ideal for problems where finding the shortest path is critical.

4.3 Question3

Findings:

- **Order of Exploratio:** UCS explores paths based on the cost of reaching each node, prioritizing nodes with lower cumulative costs.
- **Completeness:** UCS is complete for finite search spaces. It guarantees finding the lowest-cost solution.
- **Memory Usage:** UCS needs to store information about the cumulative cost of each explored node. That's why it requires significant memory if there are many paths.
- **Cost Optimality:** UCS guarantees finding the path with lowest-cost to the goal, which makes it an optimal search algorithm in terms of path cost.

Insights:

- **Trade-off Between Memory and Completeness:** UCS also involves a trade-off between memory usage and completeness.
- **Optimal Solution:** UCS is an optimal solution in terms of cost of the path.
- **Applicable Scenarios:** UCS excels in scenarios where finding the lowest-cost solution is crucial.

5 Challenges

The tasks of this lab required understanding and a review of the concepts related to DFS, BFS and UCS.

Task 1 was demonstrated by our instructor himself. Task 2 was very similar to Task 1 in terms of implementation. For Task 2, we needed to follow the same steps as task 1 with a queue.

However, Task 3 seemed a bit complex to me at first. It took me some time to understand how the priority queue was implemented in util.py. Understanding the way of using util.py's implementation was a challenge for me.

I was unsure how to extract the cost of an action at first. Later I was able to rightly call and execute the getCostOfActions() method.