Islamic University of Technology

Report on Lab 01

(CSE 4618 Artificial Intelligence Lab)

Name:	**Md Mahmudul Islam Mahin**
Student ID:	**210042140**
Department:	CSE
Program:	BSc in SWE
Date:	9^{th} June 2025

Contents

1 Introduction

In this lab, our Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. We will build general search algorithms and apply them to Pacman scenarios.

2 Problem Analysis

Pacman lives in a shiny blue world of twisting corridors and tasty round treats. Navigating this world efficiently will be Pacman's first step in mastering his domain.
In ***searchAgents.py***, we 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 is our job to implement.

3 Solution Explanation

DFS

DFS goes deep into the maze before checking sideways paths.

- It tries one path all the way to the end.
- If that fails, it backtracks and tries other options.

```python
def depthFirstSearch(problem):

    from util import Stack

    stack = Stack()
    visited = set()

    stack.push((problem.getStartState(), []))

    while not stack.isEmpty():
        current_state, path = stack.pop()

        if problem.isGoalState(current_state):
            return path

        if current_state not in visited:
            visited.add(current_state)
            for successor, action, stepCost in problem.getSuccessors(current_state
    ):
                if successor not in visited:
                    stack.push((successor, path + [action]))

    return []
    util.raiseNotDefined()
```

BFS

BFS explores all nodes at the current level first, ensuring the shortest path is found.

- It finds the shortest number of moves to the goal.
- Always explores the shallowest node first.

```python
def breadthFirstSearch(problem):

    from util import Queue

    queue = Queue()
    visited = set()

    queue.push((problem.getStartState(), []))

    while not queue.isEmpty():
        current_state, path = queue.pop()

        if problem.isGoalState(current_state):
            return path

        if current_state not in visited:
            visited.add(current_state)
            for successor, action, stepCost in problem.getSuccessors(current_state
    ):
                if successor not in visited:
                    queue.push((successor, path + [action]))

    return []

    util.raiseNotDefined()
```

UCS

It picks the path with lowest total cost from start to that node at each step.

- Expands the cheapest path so far.
- Guaranteed to find the least total cost path.

```python
def uniformCostSearch(problem):

    from util import PriorityQueue

    pq = PriorityQueue()
    visited = set()

    pq.push((problem.getStartState(), [], 0), 0)

    while not pq.isEmpty():
        current_state, path, cost = pq.pop()

```

```
        if problem.isGoalState(current_state):
            return path

        if current_state not in visited:
            visited.add(current_state)
            for successor, action, stepCost in problem.getSuccessors(current_state):
                if successor not in visited:
                    new_cost = cost + stepCost
                    pq.push((successor, path + [action], new_cost), new_cost)

    return []

    util.raiseNotDefined()
```

4 Findings and Insights

DFS

Data Structure

- Stack() -> LIFO – Last In, First Out

Complexity

b is branching factor, m is max depth of tree.

- **Space Complexity** O(bm);
- **Time Complexity** O(b^m);

Pros & Cons

- **Pros**
  - Low memory usage.
  - Can be faster if the solution is deep in the tree.
- **Cons**
  - Not guaranteed to find the shortest or cheapest path.
  - Can get stuck in deep or infinite paths without solution.

BFS

Data Structure

- Queue() -> FIFO – First In, First Out

Complexity

d is the depth of the shallowest goal.

- **Space Complexity** O(b^d);
- **Time Complexity** O(b^d);

Pros & Cons

- **Pros**
  - Guaranteed to find the shortest path (if all steps cost the same).
  - Good for unweighted problems (e.g., maze without different costs).
- **Cons**
  - Can use a lot of memory.
  - Slower than DFS for deep solutions.

UCS

Data Structure

- PriorityQueue()

Complexity

c is the cost of the cheapest solution (can vary)

- **Space Complexity** O(b^c)
- **Time Complexity** O(b^c)

Pros & Cons

- **Pros**
  - Guaranteed to find the least-cost path.
  - Works well with weighted graphs.
- **Cons**
  - Slower than BFS/DFS if all costs are equal.
  - Can also use a lot of memory.

5 Challenges Faced

- I couldn't enter the conda environment as during installation I forgot to checkmark "Add files to the directory"
- The ***autograder.py*** was only grading the first question as I didn't run each function individually at first.

6 Additional Information

After completing the task, I ran the ***autograder.py*** and it had shown the following output:

```
Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
------------------
Total: 9/9
```

7 GitHub Repository

I will be uploading the lab tasks in the following repository: CSE 4618: Artificial Intelligence