Islamic University of Technology

Report on Lab 02

(CSE 4618 Artificial Intelligence Lab)

Name:	**Md Mahmudul Islam Mahin**
Student ID:	**210042140**
Department:	CSE
Program:	BSc in SWE
Date:	20^{th} July 2025

Contents

1 Introduction

This lab focused on implementing and analyzing informed search algorithms. The tasks included:

- Implementing the A* search algorithm.
- Defining and solving the CornersProblem, where the agent visits all four corners of a grid world, using various search algorithms.
- Designing heuristics for the CornersProblem and FoodSearchProblem
- Designing a replanning agent, ClosestDotSearchAgent, and solving the AnyFoodSearchProblem, to find the closest dot in the grid world.

The following sections provide detailed discussions of the algorithms' implementations, explanations, and analyses, along with observations and experimental results.

2 Task 01: A* Search

2.1 Solution

In terms of implementation, A* search is similar to Uniform Cost Search with the only difference being the priority queue. We have implemented the UCS algorithm in the previous lab, so I am not going through the whole implementation of the A* search algorithm to avoid redundancy, only mentioning the key difference. This time the priority queue is based on the sum of the actions' cost and the state's heuristic value. So, in the implementation, I needed to change only one line:

```
fringe = util.PriorityQueueWithFunction (lambda x:problem.getCostOfActions(x[1]) +
heuristic (x [0] , problem )) # x --> (state , actions )
```

The rest of the implementation is the same as the Uniform Cost Search algorithm.

2.2 Explanation

For any node n, if the cost to reach the node is g(n) and the heuristic value is h(n), then the priority of the node is f(n) = g(n) + h(n). So, the A* search function takes the problem and the heuristic function as input. The PriorityQueueWithFunction determines the priority of the nodes based cost of the actions and the heuristic value. The node with the lowest priority is expanded first

2.3 Analysis

Time Complexity: The time complexity of the A* search algorithm is O(b^d) , where b is the branching factor and d is the depth of the optimal solution.

Space Complexity: The space complexity of the A* search algorithm is O(b^d) , where b is the branching factor and d is the depth of the optimal solution.

Completeness: The A* search algorithm is complete if the branching factor is finite and the cost of the actions is positive.

Optimality: The A* search algorithm is optimal if the heuristic is admissible in tree search and consistent in graph search, given that the edge costs are positive.

Comparison with Uniform Cost Search: The advantage of the A* search algorithm over the Uniform Cost Search is that it begins the search with some known information about the problem, an estimation, which helps to find the goal node faster. The A* search algorithm is more efficient than the Uniform Cost Search algorithm. In our Pacman game project, if we run the game for the bigMaze layout with the A* search algorithm, it finds the goal faster than the Uniform Cost Search algorithm. A* expands 549 nodes, whereas the Uniform Cost Search algorithm expands 620 nodes.

3 Task 02: Finding All the Corners

In this task, we had to define a new search problem called CornersProblem. The problem is to visit all four corners of a grid world, the agent can move in four directions — North, South, East, and West.

3.1 Solution

We have implemented the CornersProblem class in the searchAgents.py file. We defined the properties: Please remember the following text:

State space: The state space includes the position of Pacman and the visited corners. The position of the Pacman was represented by (x, y) tuple, and a dictionary of (x, y): boolean was used to represent the visited corners. The dictionary was initialized with False for all corners. This part was implemented in the CornersProblem class constructor.

The Start State: The start state of the problem was defined as the position of the Pacman and the visited corners. The position of the Pacman was the initial position of the Pacman and the visited corners were initialized with False for all corners. This part was implemented in the getStartState() method of the CornersProblem class.

The Goal Test: The goal test of the problem was to check if all the corners were visited. So, if all the corners in the visitedCorners dictionary were True, the goal was reached. This part was implemented in the isGoalState() method of the CornersProblem class.

The Successor Function: The successor function of the problem was to return the next states from the current state. The next states were generated by moving the Pacman in all four directions — North, South, East, and West. This part was implemented in the getSuccessors() method in the CornersProblem class.

- If the next state was a wall, then the state was not added to the next states.
- If it was a corner, it was marked as visited in the next state.
- If it was neither a wall nor a corner, then the next state was added to the next states.
- The cost of the action was 1.
- The successors were returned as a list of (state, action, cost) tuples.

4 Task 03: Corners Problem: Heuristic

Our search problem was ready now. But, to facilitate the A* search, we also needed to define a heuristic function for the problem, that will be used in the A* search algorithm we implemented in task 1 (Section 2).

4.1 Solution

The heuristic function of the problem was to estimate the cost from the current state to the goal state. The method was named cornersHeuristic() and implemented outside the CornersProblem class. The heuristic value was calculated as the Manhattan distance between the current position of the Pacman and the farthest among the remaining corners.

4.2 Why Max of Manhattan Distances?

First of all, it is a consistent heuristic. We need at least that many moves to reach the goal. But before that, I tried some other heuristics. Those heuristics are discussed in Observations (Section 7). If we have two heuristics, h1 and h2 , then the max of the two heuristics, say , is admissible if both h1 and h2 are admissible. It is also consistent if both h1 and h2 are consistent. The closer the heuristic value is to the actual cost, the better the heuristic is. It expands fewer nodes and finds the goal faster.

5 Task 04: Eating All the Dots

In this task, we had to design a heuristic again, this time for the FoodSearchProblem. The problem was to eat all the dots in the grid world.

5.1 Solution

Just like the CornersProblem heuristic (Section 4), I have tried the same heuristics like, the number of remaining dots, the sum of the Manhattan distances between the current position and all the remaining dots, and the Manhattan distance between the current position and the nearest dot. But none of them were good enough.

6 Task 05: Suboptimal Search

This task was about a replanning agent. Instead of planning the whole path at once, the agent plans the path to the next goal after reaching the current goal. The ultimate plan for this problem turns out to be suboptimal. But it finds the goal faster. In the task, the agent was ClosestDotSearchAgent, implemented, except the findPathToClosestDot() method. Also, this agent calls for the AnyFoodSearchProblem class, which was also unimplemented.

6.1 Solution

For the AnyFoodSearchProblem class, the goal is to eat any dot in a maze. So the goal test for this is: If the current position of the Pacman is a dot, then the goal is reached. Return True in that case. Otherwise, return False. Now the ClosestDotSearchAgent solves the AnyFoodSearchProblem problem. In the findPathToClosestDot() method, we call one of the search algorithms to find the path to the closest dot, and return the path segment.

6.2 Explanation

The ClosestDotSearchAgent is a replanning agent that plans the path to the next goal after reaching the current goal. It continues this process until all the dots are eaten. In each iteration, the agent:

- Registers a new start state, which is the current position of Pacman and the remaining food dots.
- Instantiates the AnyFoodSearchProblem class with this new start state.
- Calls a search algorithm to find the path to the closest dot.

When Pacman reaches a dot, the AnyFoodSearchProblem goal test returns True, and the agent registers a new start state. It then finds the path to the next closest dot. This process continues until all the dots are eaten. In summary, the ClosestDotSearchAgent repeatedly formulates and solves the AnyFoodSearchProblem in each iteration until all the dots are consumed.

6.3 Analysis

The ClosestDotSearchAgent is a replanning agent, it does not garanty the optimal path. But it solves the problem faster than the optimal path.

7 Conclusion

In this lab, we explored informed search algorithms and their application to the Pacman problem. We implemented and tested heuristics for the CornersProblem and AnyFoodSearchProblem, analyzing their effectiveness. Key takeaways include:

- The importance of choosing appropriate heuristics for informed search algorithms.
- The Manhattan distance to the farthest dot heuristic was the most effective for the CornersProblem.
- The ClosestDotSearchAgent demonstrated the trade-offs between optimality and computational efficiency.

8 GitHub Repository

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