Submitted By- 200041125

Mehedi Ahamed

Introduction

This report provides an overview of the implementation and analysis of Informed Search algorithm specially A* search and appropriate heuristics for various Pacman scenarios. The goal is to achieve a perfect score from the autograder file, indicating the successful completion of all assigned challenges.

Problem Analysis (A* Search)

To implement the A* Graph Search algorithm, I needed to complete the function aStarSearch in the search.py file. The A* algorithm utilizes a heuristic function, which takes two parameters:

```
def aStarSearch(problem, heuristic=nullHeuristic)
```

For the initial task of navigating a maze to reach a specific target location, I applied the A* algorithm using the Manhattan distance as the heuristic.

Solution Approach (A* Search)

In this solution, the A* search algorithm is implemented to find the optimal path from the start state to the goal state in a problem. The approach begins by initializing a priority queue, called the fringe, which will store nodes to explore. The queue is ordered by the sum of the cumulative cost to reach a node and a heuristic estimate of the cost to reach the goal from that node. The algorithm also initializes a list to keep track of visited states. The search starts from the root node, which is the initial state of the problem, with a cost of zero. The node is then pushed into the fringe. The algorithm proceeds by continuously popping nodes from the fringe, checking if the current state is the goal. If the goal is reached, the plan (or list of actions) that leads to the goal is returned. For each state that hasn't been visited yet, the algorithm explores its successors (possible next states). It adds these successors to the fringe with an updated cost and an extended plan of actions. The cost is calculated by adding the cumulative cost to reach the current node, the cost of the action to the successor, and the heuristic estimate for the successor state. The algorithm ensures that nodes are visited only once by adding the current state to the visited list after exploring it. The process repeats until the goal is found or the fringe is exhausted.

By default, the heuristic is configured as a null Heuristic with a value of 0, but it can be adjusted to a more suitable heuristic.

Findings and Insights

Time Complexity: $O(b^s)$
In the worst case, the algorithm explores all nodes up to a depth of s. Here, b represents the branching factor, and s is the depth where the nearest solution is found, for a Null Heuristic.

Space Complexity: $O(b^s)$
In the worst scenario, the fringe stores nearly all the states at level s (the level containing the shallowest solution), for a Null Heuristic.

Problem Analysis (Finding All the Corners)

In corner mazes, there are four specific points—one in each corner. This new search problem requires finding the shortest path through the maze that passes through all four corners, regardless of whether there is food located there. The objective is to implement the **CornersProblem** search problem in the **searchAgents.py** file.

Solution Approach (Finding All the Corners)

1.Initialization (__init__)

The **__init__** method initializes the problem by setting up key attributes, such as the maze walls, Pacman's starting position, and the corners of the maze. The maze boundaries are stored in **self.walls**, and Pacman's initial location is recorded in **self.startingPosition**. The corners are defined based on the maze's dimensions, with coordinates for the bottom-left, top-left, bottom-right, and top-right corners. The code then checks if there is food at each corner; if not, it displays a warning message. Additionally, **self._expanded** is initialized to keep track of the number of nodes expanded during the search, which helps evaluate search efficiency. Finally, **self.visitedCorners** is set up as an empty list to track which corners Pacman has already visited.

2.Obtaining Start State

The **getStartState** method defines the starting state for the search problem in the customized state space rather than the full Pacman state space. Instead of only returning Pacman's starting position, it returns a tuple containing two elements: **self.startingPosition** (the coordinates where Pacman starts) and **tuple(self.visitedCorners)**. The **visitedCorners** is converted to a tuple to make it immutable, which is useful for tracking the visited corners without altering them throughout the search. This combined representation of the starting position and visited corners allows the search

algorithm to keep track of both Pacman's position and which corners have been visited, enabling it to plan the path efficiently.

3.Getting Successor

The **getSuccessors** method generates the possible next states (successors) for a given state in the maze. Starting with the current position (**currPosition**) and visited corners (**currVisitedCorners**), it checks four possible moves: North, South, East, and West. For each action, it calculates the new position **(nx, ny)** by adjusting the current position using direction vectors. If this new position is not a wall (**hitsWall == False**), it creates a successor. If this new position is a corner and hasn't been visited, it adds it to the list of visited corners. Then, it defines **nextState** as a tuple containing the new position and the updated visited corners. Each successor is added to the successors list with a cost of 1. Finally, it increments **_expanded** (to track expanded nodes) and returns the list of successors.

4.isGoalState

The **isGoalState** method checks if the current state is a goal by examining **currVisitedCorners**. If the number of visited corners is 4, meaning all four corners have been visited, it returns True, indicating the goal has been reached. Otherwise, it returns False.

Findings and Insights

UCS proved to be more efficient in terms of path cost compared to both BFS and DFS.

Problem Analysis (Corner Heuristic)

Implement a non-trivial consistent heuristic for the **CornersProblem** in cornersHeuristic.

Solution Approach (Corner Heuristic)

The **cornersHeuristic** function provided is designed for the **CornersProblem** where the objective is to find the shortest path to visit all corners of a maze. This function computes an admissible heuristic by calculating the distance from the current position to each unvisited corner using the Manhattan distance, which simply sums the absolute differences in the horizontal and vertical coordinates. The heuristic value returned by the function is the maximum of these distances, ensuring that it underestimates the true shortest path distance to the furthest unvisited corner. This approach

satisfies the conditions for admissibility (never overestimating the true cost to reach the goal) and helps guide the search algorithm efficiently towards the solution by focusing on the farthest point yet to be visited.

Findings and Insights

Manhattan Distance = $|x_1 - x_2| + |y_1 - y_2|$

Euclidean Distance = $\{ (x_1 - x_2)^2 + (y_1 - y_2)^2 \}^{0.5}$

Maze Distance = The function calculates the shortest path between two points, considering walls, using breadth-first search (BFS), which is precise but computationally intensive. Although the Euclidean distance is a smaller estimate than the Manhattan distance, the latter often performs better for the Pacman game since it reflects the actual movement restrictions (horizontal or vertical only, no diagonal paths). Consequently, the Manhattan distance, despite being a less direct estimate than **mazeDistance**, can often result in fewer node expansions and reduced time complexity, making it more efficient in scenarios where the BFS computation of **mazeDistance** becomes overly complex due to the maze's layout.

Problem Analysis (Eating All the Dots)

Implement a heuristic for the **FoodSearchProblem** in **foodHeuristic** such that the search agent eats all the food.

Solution Approach (Eating All the Dots)

The **foodHeuristic** function in the context of a Pac-Man game uses a heuristic based on the Manhattan distance to assess the cost of reaching each food item on the grid from Pac-Man's current position. The function iterates through all food locations, calculates the Manhattan distance to each, and stores these distances in a list. The heuristic value is determined by the maximum

distance in this list, ensuring that the heuristic remains admissible (i.e., it never overestimates the true cost) and consistent (i.e., the heuristic difference between two states will not exceed the step cost between those states). This approach helps A* search prioritize paths that move towards the farthest food item, aiming to find an efficient route to clear all food items from the grid.

Findings and Insights

mazeDistance as heuristic consumes more time but expands a lesser number of nodes compared to Manhattan or Euclidean as heuristic.

Problem Analysis (Suboptimal)

ClosestDotSearchAgent is implemented in searchAgents.py, but it's missing a key function that finds a path to the closest dot.

Solution Approach (Suboptimal)

The function **findPathToClosestDot** is designed to compute the shortest path to the nearest dot ("food") from the current position of Pacman in the game. Starting with the given **gameState**, the function retrieves the current position of Pacman, the locations of all food items, and the placements of walls which act as obstacles. It then creates a problem instance specific to finding any food using the current game state. To solve this problem and find the path to the closest dot, the function employs the Uniform Cost Search (UCS) algorithm through **search.ucs(problem)**. UCS is effective in this scenario because it expands nodes in increasing order of their path cost, thereby ensuring that the first path found to any dot is the shortest, taking into account all obstacles and the layout of the game map.

Challenges Faced

Scoring full marks in autograder was medium difficult

All the Codes are available in the below links-
Task