Problem Statement 1

Analysis:

The problem statement at hand involves implementing a search algorithm, namely A* Search Algorithm so that pathfinding problems can be solved in Pacman's World. The goal is to help Pacman find optimal paths through a maze so that it can reach a particular goal efficiently.

Explanation:

What makes A* algorithm different from BFS, DFS, UCS algorithms is that it not only considers the cost of reaching a particular state but also it considers a heuristic estimate of the remaining distance to the goal.

Implementation Logic: While exploring the nodes, A* prioritize nodes that have a lower combination of accumulated cost from the start node and a heuristic estimate of the cost to reach the goal node, leading to the efficient discovery of optimal paths for Pacman.

How the Algorithm works: This algorithm maintains a priority queue as its fringe. At each step, it selects the node with the lowest sum of accumulated cost to reach that particular node and the heuristic value of that node. This allows us to find efficient paths to the goal, since the algorithm is selecting the node that has the least F value (F Value = Backward Cost + Heuristic Cost).

After initializing the fringe (Priority Queue) with the start state and its path cost and heuristic value, in a loop, nodes are popped from the fringe and then expanded. If a goal state is reached, the algorithm returns the list of actions leading up to that goal state. Otherwise, successors of the non-goal (current) state are explored, and their costs and heuristic values are updated accordingly before pushing them into the fringe.

So, since the algorithm is selecting the node that has the least F value, this approach guides Pacman towards the goal while ensuring optimality by considering all possible paths and selecting the one that has the lowest overall cost.

Challenges:

Since the A* algorithm also considers a heuristic estimate, I had to implement it in such a way that the heuristic values are being handled properly and the F value (F Value = Backward Cost + Heuristic Cost) is also being calculated accurately – This was a bit difficult to implement.

To solve this issue, I am pushing a tuple inside the priority queue – The tuple consists of the heuristic value of the node and the node itself and for the F value, it's being calculated inside a loop for all the successors of a node. Adding the accumulated cost of the successor node with the heuristic value of it to get the F value.

Findings:

A* considers both the accumulated cost of all the actions up to a particular state and a heuristic estimate of the remaining cost to a goal state.

The implementation uses a priority queue, that ensures that nodes are explored as per lower total estimated cost (F value). This helps us find possible optimal paths.

The "nullHeuristic" returns 0 as its heuristic value. So, if we use this as our heuristic estimate, our A* algorithm gets reduced to an UCS algorithm. Basically, in this case, A* would work like UCS.

Now, if we come up with a heuristic function that is admissible, meaning that the estimation would be an understatement of the true cost, then this algorithm would lead to optimal solutions.

Coming up with a more consistent heuristic would allow this algorithm to be more robust, faster converging. But with inconsistent heuristics or less accurate heuristics would result in suboptimal solutions, slower convergence.

Problem Statement 2

Analysis:

The objective is to find the shortest path through a maze that touches all four corners. It doesn't matter if there's any food or not. A search problem needs to be implemented so that Pacman can navigate through a maze efficiently to visit all four corners.

Explanation:

The implementation logic focuses around defining a problem space so that Pacman can move through a maze to touch all four corners. This was done for us.

Implementation Logic: First, the state space (node) needs to be defined. Which is represented as a tuple containing Pacman's current position and a dictionary is maintained – indicating which corners have been visited. Then, the goal state is the state where all of the corners have been visited by Pacman.

From a particular state, Pacman has four possible actions, which are North, South, East, West. Here, we also have to consider if there are walls or not. Basically, we have to look for all the valid movements – these will be the successor states of the current state. If a valid successor state is found, we update the Pacman's position. Now, if the next position is a corner (if the Pacman moves to a corner), we mark it as visited. This is how – by exploring successor states, we can make sure Pacman visits all four corners eventually.

Overall, the implementation applies search algorithms to the problem at hand by representing the proper state space(s), defining the goal state, generating valid successor states.

Challenges:

It took me a bit of time to understand the already written/implemented program code for this problem.

Findings:

The cost of action is set to 1 (same for all actions). So, we don't have to consider the possibility of some actions costing more than usual or the costs being random. If the cost was not same for all actions, then we won't be able to find optimal solutions, since we are relying on BFS.

The implementation keeps track the progress of the search by representing states as tuples containing Pacman's position and a dictionary of visited corners. The successor function ensures that only valid states are explored.

Here, there are only four actions available for Pacman. More available actions would result into more nodes being explored and a search algorithm would take longer than usual to find all of the corners.

Problem Statement 3

Analysis:

The problem statement at hand involves implementing a non-trivial, consistent heuristic for the corners problem. We have to implement a heuristic that is an underestimate of the actual cost – we have to find a lower bound on the shortest path from current state to a goal state (all four corners visited). The heuristic has to be admissible and consistent (almost all admissible heuristics will be consistent as well).

Explanation:

The implementation logic revolves around identifying the farthest unvisited corner from a particular position and returning its Manhattan value as heuristic value – this estimation ensures admissibility and consistency of the heuristic function.

We are taking the farthest corner rather than any other corners, because the value that we get considering the farthest corner will give us the heuristic value that is the closest to the actual cost and we all know that the closer heuristic value is to the actual cost, the better the algorithm performs. Results in lesser nodes being expanded. That is why we will take the maximum of all the heuristic estimations.

Besides we are selecting the Manhattan distance here, which helps to ensure that the heuristic is a lower bound on the shortest path from the current state to a goal state.

In terms of the implementation logic, first, we extract the current position and the visited corners. After determining the unvisited corners, we have to figure out the farthest unvisited corner. For each corner we have to calculate the Manhattan distance from the current position, from that we will extract the maximum value. The maximum value in this case (farthest unvisited corner) will be the heuristic value. This is how we can come up with a non-trivial consistent heuristic for the corners problem.

Challenges:

I tried to use the Manhattan heuristic function that was already implemented. But it wasn't working for me – was getting "attribute errors". Later I used the Manhattan distance function from the "util.py". Took me a while to find out that this helper function was already implemented for us.

Findings:

By considering the distance to the farthest unvisited corner, the heuristic ensures admissibility and consistency. In case of admissibility – the farthest unvisited corner gives us a value that is a lower bound on the actual shortest path cost to the nearest goal. In case of consistency, farthest unvisited corner hold that if an action has a particular cost, then taking that action can only cause a drop in heuristic of at most of that particular cost.

The implementation iterates through the unvisited corners and calculates the Manhattan distance for each, using data structures such as lists and dictionaries to track corner states.

The number of expanded nodes will most likely change if a different heuristic function (i.e., Euclidian distance) is used.

Rather than considering the farthest corner, if we consider any other corner's Manhattan distance as the heuristic value, this would result in expanding a lot more nodes.

Corner	Nodes Expanded
Farthest Unvisited Corner	1136
Any Unvisited Corner (Other than the farthest one)	1322

<u>Problem Statement 4</u>

Analysis:

The problem statement at hand involves finding a path for Pacman to collect all the food in the maze in as few steps as possible. A non-trivial, consistent heuristic needs to be implemented for the food problem so that we reach the goal optimally. Pacman's position and the remaining food grid are the states that we are concerned with. As mentioned before, the heuristic that we have to implement has to be an underestimate of the actual cost.

Explanation:

The Implementation Logic: We iterate over all the food pellets and calculating their distances. From that we extract the maximum distance. So, we identify the farthest food pallet from a particular position and return its minimum cost to reach that food pallet as the heuristic value – this estimation ensures admissibility and consistency of the heuristic function.

We are taking the farthest food pallet rather than any other food pallets, because the value that we get considering the farthest food pallet will give us the heuristic value that is the closest to the actual cost and we all know that the closer heuristic value is to the actual cost, the better the algorithm performs. Results in lesser nodes being expanded. That is why we will take the maximum of all the heuristic estimations.

Moreover, we are running BFS or UCS to get the minimum distance from a particular node (start node) to the farthest food pallet, which is also the goal node. Selecting the minimum distance here helps to ensure that the heuristic is a lower bound on the shortest path from the current state to a goal state.

We extract the current position of Pacman and the grid of remaining food pellets. Then we iterate over the remaining food pallets and calculate the minimum distance from Pacman's current position to that food pellet using a helper function (mazeDistance). The helper function (mazeDistance) calculates the shortest path distance between two points in a maze environment, it also has to consider the layout of walls while calculating the distance. In this case, the search algorithm used is BFS We could’ve also used UCS. Then the maximum distance out of all the food pallets is returned as heuristic value.

This is how this solution helps prioritize the actions that brings Pacman closer to achieving the goal of collecting all the food pellets in as few steps as possible.

Challenges:

It took me a bit of time to figure out there was a helper function that would help us find the minimum distance between Pacman's current position to a particular food pellet.

Findings:

If the food grid is empty, that means that all food has been collected, it returns a heuristic value of 0. If the grid is not empty, then the heuristic value of each node is the estimate of reaching the farthest food pallet from that node.

Here, the helper function uses BFS as the search algorithm. The number of nodes expanded will differ based on the search algorithm that was chosen. For example, DFS fails admissibility test of the heuristic.

Algorithm	Admissibility Test
BFS	Passes
DFS	Fails
UCS	Passes

DFS traverses down a single branch of the search tree until it reaches a leaf node before backtracking and exploring other possibilities. That’s why, DFS does not consider all possible paths before finding a solution, and the first solution found may not necessarily be the optimal one.

The number of nodes expanded also differs based on the strategy that we choose. Here are a few strategies and the number of nodes that gets expanded after using these:

Strategy	Nodes Expanded
Manhattan Distance for the Farthest Food Pallet	9551
Manhattan Distance for the last food pallet in the food grid list	11269
Minimum Distance for the Farthest Food Pallet **(We are using this strategy)**	4137
Minimum Distance for the last food pallet in the food grid list	8046

<u>Problem Statement 5</u>

Analysis:

The problem statement at hand involves implementing an agent that greedily eats the closest dot in the Pacman maze. This agent should be able to find a reasonably good path to the nearest dot, even if finding the optimal path is challenging. There's also a need to define a goal test function to determine whether Pacman has reached any food dot or not.

Explanation:

Implementation Logic: Find a path from the Pacman's current position to the closest food dot in the maze – create or define a search problem with the help of a helper class and then solve that search problem with the help of search algorithms that are already implemented.

the algorithm explores the maze to find the optimal path to the nearest food dot. The goal is to return a list of actions that Pacman can take to reach the closest dot, starting from the current game state. Overall, the function helps Pacman greedily eat the closest dot in the maze.

Implementation Logic: As for the goal state test, we try to figure out if the given state holds a food pallet and also it is the current position of Pacman or not. We extract the x and y coordinates from Pacman's current position and check whether a food dot is present at the same position or not. If yes, it's a goal state. Otherwise, it's not a goal state.

Challenges: No significant challenges faced.

Findings:

After defining a suitable search problem and implementing the goal test function, Pacman can now navigate the maze to reach the nearest food dot. Although the time it takes or how fast Pacman can reach the closest food dot can depend on the algorithm used. In this case, BFS, UCS, A* works just fine, but DFS fails.

DFS explores as far as possible along each branch before backtracking. It does not guarantee finding the optimal solution and may find a suboptimal path to the closest food dot. That's why DFS fails in this case.

Although the behavior of A* may vary depending on the heuristic function used.