Question - 1 : A* Search

Introduction :
In this task, we implemented A* graph Search in the empty function ***aStarSearch()*** which is defined in the ***search.py*** file. As per the function definition, aStarSearch() takes the problem and a heuristic as the function parameters.

Problem Analysis :
A* search is almost similar to the Uniform Cost Search in the implementation level but in A* search we need to add the heuristic cost with the incurred cost of the actions. So, Pacman will choose the path which has lowest summation value of heuristic cost and the actual cost in A* Search.

Solution Explanation:
As mentioned earlier, other than the heuristic addition part, the solution is mostly similar to the UCS solution that we have done in the previous lab. Here we also took the Priority Queue as the fringe and a visited[ ] array to keep track of the visited nodes so that pacman doesn't visit the same state multiple times.
The core difference here is : In UCS the priority of the priority queue was calculated only based on cumulative path cost but in A* search the priority is calculated by the sum of the cumulative path cost and heuristic estimate which makes it a better informed search algorithm.

Challenges :
As the solution approach was instructed in the lab, I have not faced any particular problem in this task.

Question - 2 : Finding all the Corners

Introduction :
In a maze having four dots (food pallets) in four different corners, we needed to find the shortest path in that maze which touches all the four corners and eat all the food pallets while avoiding touching the walls in the maze.

Problem Analysis :
As per the problem statement, to use ***breadthFirstSearch()*** function from ***search.py*** that has already been implemented, we need to formulate this shortest path finding problem as a search problem first. Along with the pre-implemented part, we just needed to define the necessary functions like ***getStartState() , isGoalState()*** and ***getSuccessors()*** to formulate this problem as a proper search problem.

Solution Explanation :
In the ***getStartState()*** function initializes a starting point combining the initial position and and the visited corners in the ***startState*** tuple. In the ***isGoalState()*** function, it returns true if all the corners are visited, otherwise it returns false.
getSuccessors() function produces next probable states, actions and costs according to the current state, direction of movement of the pacman and the obstacles (wall) in the maze while updating the visited corners accordingly.

Challenges :
As the solution of this task was directly demonstrated in the lab, I have not faced any problem in this task

Question - 3 : Corners Problem - Heuristic

Introduction:
In this problem, we needed to implement a non trivial heuristic for the CornersProblem in cornersHeuristic() function which is defined in the searchAgents.py. The Heuristic should be consistent and admissible.

Problem Analysis :
As the Corners Search Problem is being performed with A* Search, we need to provide a heuristic as the function parameter of aStarSearch(). Here in the ***cornersHeuristic (state,problem)*** **-** state is the current search state and problem is referring to the ***CornersProblem.***

Solution Explanation :
I used the max manhattan distance as the heuristic. The function for determining Manhattan distance between two points was already defined in the ***util.py***.
Here the maximum manhattan distance from the current position of Pacman to the unvisited corner in the maze gives a lower bound on the overall shortest path because the pacman cannot move diagonally in the maze and it needs to move at least as much as the manhattan distance. As this calculation never overestimates the actual cost, this heuristic is admissible. This heuristic is also consistent, since the manhattan distance updates consistently every time when the search progresses towards next corner to explore.

Challenges :
I needed to try out some probable heuristic like max/min euclidean distance, max/min manhattan distance to pass all the test cases. This is not a challenge itself, but trying out all possible heuristics to pass the test cases was a bit hectic.

Question - 4 : Eating all the Dots.

Introduction :

In this particular problem, we needed to make the Pacman eat all food dots in the maze in as few steps as possible. This problem is defined as Food Search Problem and the food search problem will use the ***aStarSearch()*** from ***search.py***. So, we needed to provide a certain heuristic which will be used as the heuristic parameter of aStarSearch. We needed to give our heuristic in the ***foodHeuristic(state, problem)*** function.

Problem Analysis :

The problem requires a consistent heuristic which is the estimated cost for Pacaman in the maze to reach a particular node from the current node. We cannot use trivial heuristics like manhattan distance here because, some of the given test cases are made complex with a number of walls in the maze for which we won't get any manhattan distance between the pacman and the food palette. So a different kind of heuristic is needed here.

Solution Explanation :

In the lab, Firstly, I tried out euclidean and manhattan distance as heuristics but they could pass only one or two test cases. But later found out, ***mazeDistance(point1, point2, gameState)*** function is defined in the ***searchAgents.py*** for the use. It firstly checks if the points are walls or not and then formulates a Search problem and uses BFS to find the shortest path (least node expanded) between the points while avoiding the walls also. Here in the return statement, the length refers to the number of nodes expanded while performing BFS.

So I used mazeDistance() function to formulate the heuristic to get the closest food packet from the pacman's position.

```
position, foodGrid = state
positionsOfPackets = foodGrid.asList()

if not positionsOfPackets:
    return 0  # No remaining food, heuristic is 0

closest_food = mazeDistance(position, positionsOfPackets[0], problem.startingGameState)
return closest_food
```

Challenges :

Initially I didn't notice the mazeDistance was given for use. I tried out some probable heuristics but couldn't pass all the test cases with them. Then I found out the reason behind and tried some custom heuristics also. At the end tried mazeDistance and got the result, but putting proper arguments into the mazeDistance() needed some tinkering also in which I faced difficulties in deciding which state to use as the gameState.

Question - 5 : Suboptimal Search

Introduction :
We needed to implement the incomplete ***findPathToClosestDot()*** function under ClosestDotSearchAgent class in this problem.

Problem Analysis :
The problem requires defining ***findPathToClosestDot()*** such as the pacman always greedily eats the closest dot. So we need to use that kind of Search Algorithm from search.py which cannot guarantee the most cost effective path always but can give the shortest path to an intermediate goal.

Solution Explanation :
I simply used the ***bfs()*** function from the ***search.py*** and passed the problem as the argument into it. We know that, BFS doesn't prioritize the cost effective path always but it can guarantee the shortest path to a goal.
Because of this lack of consideration for varying path cost, BFS becomes the suboptimal solution for this ClosestDotSearchAgent problem while A* Star Search or UCS can be the optimal solution.

Challenges :
I have not faced any particular challenges in this problem as this problem was pretty straightforward and intuitive.