ISLAMIC UNIVERSITY OF TECHNOLOGY

Organization of Islamic Cooperation

Board Bazar, Gazipur

CSE 4712: Artificial Intelligence Lab

Report: Lab 2

Name: Tasfia Tasneem Annesha

Student Id: 190041220

Lab group: 2B

Task 1:

Question explain:
Task 1 is A* search algorithm.

Solution:

```
def aStarSearch(problem, heuristic=nullHeuristic):
"""Search the node that has the lowest combined cost and heuristic
first."""
    fringe = util.PriorityQueue()
    visited = []
    fringe.push((problem.getStartState(), []), 0)
    while not fringe.isEmpty():
        currState, actionList = fringe.pop()
        if problem.isGoalState(currState):
            return actionList
        if currState not in visited:
            for nextState, action, cost in
problem.getSuccessors(currState):
                fringe.push((nextState, actionList +
[action]),problem.getCostOfActions(actionList + [action]) +
heuristic(nextState,problem))
            visited.append(currState)
    return None
```

Code explain:
This A* search is similar to the uniform cost search that we have done in the previous lab. Here the fringe is also a priority queue. But here the priority queue is defined by the summation of the actual cost + the heuristic cost for reaching a particular node.

Task 2:

Question explain:
The second ques is about Corner Problem where pacman needs to eat the 4 dots in each corner of the maze.

Solution explain:
Task 2 was done by sir in the class. For this at first we define _init_ function. To keep track of whether or not the dots of the corners have been consumed, we define a dictionary of coordinates as the keys and booleans as the values. At first, they are all set to "false." However, the corresponding boolean is set to true once

a specific corner has been reached.By determining whether each corner has a true value, the goal state is determined.

```
    def __init__(self, startingGameState):
        """
        Stores the walls, pacman's starting position and corners.
        """
        self.walls = startingGameState.getWalls()
        self.startingPosition = startingGameState.getPacmanPosition()
        top, right = self.walls.height-2, self.walls.width-2
        self.corners = ((1,1), (1,top), (right, 1), (right, top))
        for corner in self.corners:
            if not startingGameState.hasFood(*corner):
                print('Warning: no food in corner ' + str(corner))
        self._expanded = 0 # DO NOT CHANGE; Number of search nodes expanded
        # Please add any code here which you would like to use
        # in initializing the problem
        "*** YOUR CODE HERE ***"
        self.visitedCorners = {}
        for corner in self.corners:
            self.visitedCorners[corner] = False

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman
state
        space)
        """
        "*** YOUR CODE HERE ***"
        # util.raiseNotDefined()
        return (self.startingPosition, self.visitedCorners)


    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        # util.raiseNotDefined()
        currPos, currVisitedCorners = state
        for corner in self.corners:
            if currVisitedCorners[corner] == False:
                return False

        return True
```

In the getSuccessors function we analyze the state that results from each possible action. We move to that state if the action does not result in a wall, and if this state is a corner, we set it to true. The function appends the "successor" list with the next state, the most recent "visited" dictionary, the action to take to get there, and its cost before returning it.

```
    def getSuccessors(self, state):
        """
        Returns successor states, the actions they require, and a cost of 1.

         As noted in search.py:
            For a given state, this should return a list of triples,
(successor,
            action, stepCost), where 'successor' is a successor to the
current
            state, 'action' is the action required to get there, and
'stepCost'
            is the incremental cost of expanding to that successor
        """

        successors = []
        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST,
Directions.WEST]:
            # Add a successor state to the successor list if the action is
legal
            # Here's a code snippet for figuring out whether a new position
hits a wall:
            #   x,y = currentPosition
            #   dx, dy = Actions.directionToVector(action)
            #   nextx, nexty = int(x + dx), int(y + dy)
            #   hitsWall = self.walls[nextx][nexty]

            "*** YOUR CODE HERE ***"
            currPos, currVisitedCorners = state
            currX, currY = currPos
            dx, dy = Actions().directionToVector(action)
            nextX, nextY = int(currX + dx), int(currY + dy)
            hitsWall = self.walls[nextX][nextY]

            if not hitsWall:
                nextPos = (nextX, nextY)
                nextVisitedCorners = deepcopy(currVisitedCorners)
                if nextPos in self.corners:
```

```
                    nextVisitedCorners[nextPos] = True

                nextState = ((nextPos, nextVisitedCorners), action, 1)
                successors.append(nextState)



        self._expanded += 1 # DO NOT CHANGE
        return successors
```

Task 3:

Ques explain:
This task is to design a heuristic function for the corner problem. The heuristic must be both consistent and admissible. The Manhattan Distance metric is the obvious solution to such a requirement, and it yields a successful outcome.

```
def cornersHeuristic(state, problem):
    """
    A heuristic for the CornersProblem that you defined.

      state:   The current search state
               (a data structure you chose in your search problem)

      problem: The CornersProblem instance for this layout.

    This function should always return a number that is a lower bound
on the
    shortest path from the state to a goal of the problem; i.e.  it
should be
    admissible (as well as consistent).
    """
    corners = problem.corners # These are the corner coordinates
    walls = problem.walls # These are the walls of the maze, as a Grid
(game.py)

    "*** YOUR CODE HERE ***"

    heuristic = [0]
    pos, currVis = state
    for corner in corners:
```

```
        if currVis[corner] is False:
            cost = util.manhattanDistance(pos, corner)
            heuristic.append(cost)
    return max(heuristic)
```

Explanation:

The list of heuristic costs is initially initialized as a list with a 0 added to it. Then, starting from our current location, we calculate the manhattan distance between all of the unexplored corners. The total of all the distances will be the heuristic cost for this problem. Why we choose the greatest of all possible distances is a concern with this method of solution. We must first describe how the heuristic cost is calculated in order to respond to that. In the Corners Problem, the heuristic cost calculates the distance to a potential goal state. The farthest corner is designated as a provisional goal because there isn't a fixed goal state. The goal state would be reached if you arrived at this corner, effectively ending the game. The heuristic cost to reach this provisional goal state is therefore indicated by returning the maximum Manhattan distance to any corner.

The idea behind this heuristic is that it provides a lower bound on the cost to visit all unvisited corners from the current state. The heuristic ensures that it is at least as costly to reach any unvisited corner by taking the highest cost among all unexplored corners. This heuristic is acceptable because it makes the assumption that all corners can be reached directly without taking into account any obstacles (walls), so it never overestimates the true cost. Additionally, it makes sense because the price to get to any corner can only get cheaper as the search goes on.

Task 4:

Ques explain:

The question asked for the implementation of a consistent heuristic for the FoodSearchProblem. This heuristic should be admissible and work with a state represented as a tuple containing the Pacman's position and a grid of food locations. The heuristic can also access additional information like walls and store reusable data in the problem.heuristicInfo dictionary.

```
def foodHeuristic(state, problem):
    """
    Your heuristic for the FoodSearchProblem goes here.
```

```
    This heuristic must be consistent to ensure correctness.  First,
try to come
    up with an admissible heuristic; almost all admissible heuristics
will be
    consistent as well.

    If using A* ever finds a solution that is worse uniform cost search
finds,
    your heuristic is *not* consistent, and probably not admissible!
On the
    other hand, inadmissible or inconsistent heuristics may find
optimal
    solutions, so be careful.

    The state is a tuple ( pacmanPosition, foodGrid ) where foodGrid is
a Grid
    (see game.py) of either True or False. You can call
foodGrid.asList() to get
    a list of food coordinates instead.

    If you want access to info like walls, capsules, etc., you can
query the
    problem.  For example, problem.walls gives you a Grid of where the
walls
    are.

    If you want to *store* information to be reused in other calls to
the
    heuristic, there is a dictionary called problem.heuristicInfo that
you can
    use. For example, if you only want to count the walls once and
store that
    value, try: problem.heuristicInfo['wallCount'] =
problem.walls.count()
    Subsequent calls to this heuristic can access
    problem.heuristicInfo['wallCount']
    """
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    foodCoords = foodGrid.asList()
    heuristic = [0]
    for foodCoord in foodCoords:
```

```
        cost = mazeDistance(position,
foodCoord,problem.startingGameState)
        heuristic.append(cost)
    return max(heuristic)
```

Code explain:
This `foodHeuristic` function calculates a heuristic for the FoodSearchProblem. It estimates the cost to reach the farthest food pellet from the current Pacman position. It computes the distance to each food pellet using `mazeDistance`, which measures the number of moves required in the game's maze to reach the target. The heuristic returns the maximum distance, ensuring it's admissible as it never overestimates the true cost and consistent because moving towards any food pellet can only reduce the distance to it as the search progresses.This gives the optimal path for this problem.

Task 5:

Ques explain:
Finding the right plan of action is not always a simple task, according to the thesis of question 5. Instead, we might prefer to quickly locate a reasonably good path. Therefore, we had to write code to make the agent eat the nearest food with greed.

```
    def findPathToClosestDot(self, gameState):
        """
        Returns a path (a list of actions) to the closest dot, starting
from
        gameState.
        """
        # Here are some useful elements of the startState
        startPosition = gameState.getPacmanPosition()
        food = gameState.getFood()
        walls = gameState.getWalls()
        problem = AnyFoodSearchProblem(gameState)

        "*** YOUR CODE HERE ***"
        #util.raiseNotDefined()
        def greedyHeuristic(position, foodGrid):
            foodCoords = foodGrid.asList()
            heuristic = [0]
            for foodCoord in foodCoords:
                cost = util.manhattanDistance(position, foodCoord)
```

```
                heuristic.append(cost)
            return min(heuristic)
        fringe = util.PriorityQueue()
        visited = []
        fringe.push((problem.getStartState(), []), 0)
        while not fringe.isEmpty():
            currState, actionList = fringe.pop()
            if problem.isGoalState(currState):
                return actionList
            if currState not in visited:
                for nextState, action, cost in
problem.getSuccessors(currState):fringe.push((nextState, actionList +
[action]),greedyHeuristic(startPosition, food))
                visited.append(currState)
        return None
```

Solution explain:
The chosen approach is to employ a greedy search strategy, closely resembling Uniform Cost Search (UCS). However, the key difference lies in the priority calculation, which solely relies on the heuristic cost. In this context, the heuristic is defined as the minimum Manhattan distance among all food pellets. This approach has yielded the highest score for this particular problem.

Code explain:
It starts by extracting relevant information from the `gameState`, including the Pacman's starting position, the layout of food, and maze walls.Here I take a heuristic function called `greedyHeuristic`, which estimates the distance from a given position to the nearest food pellet using Manhattan distance.The code sets up a priority queue called `fringe` and a list called `visited` to keep track of visited states. It initializes the fringe with the start state and an empty list of actions. Then it performs a search by expanding states in the fringe while considering the closest food pellet as the goal. It uses the `greedyHeuristic` to prioritize states with lower heuristic values, effectively guiding Pacman to the nearest food. If a goal state is found (Pacman reaches the closest food pellet), the function returns the list of actions leading to that state. If no path is found, it returns `None`. This approach ensures that Pacman greedily chooses the closest food pellet while searching efficiently.

—--------------------------------X---------------------------------
--