Lab 2

Tasnimul Hasnat

190041113

Wednesday, Sep 13 2023

Task 1

```
    def aStarSearch(problem, heuristic=nullHeuristic):
        fringe = util.PriorityQueue()
        visited = []
        fringe.push((problem.getStartState(), []), 0)

        while True:
            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)
```

A* search is the same as Uniform Cost search (UCS) except for a few things. I use priority queue here as the fringe is the same as UCS. The only difference here is in the aspect of priority. The priority here is defined as the sum of the actual cost and the heuristic (estimated) cost to reach a specific node. The heuristic cost is calculated in a function called heuristic() which takes the next state and the problem as parameters. For the start state the priority is set to 0. Then when I push the successor states to the fringe, the priority is calculated and pushed with the state. This function will return the optimal path with the help of the defined priority.

Task 2

```
def __init__(self, startingGameState):

            ...  ...  ...  ...  ...
    self.visitedCorners = {}
    for corner in  self.corners:
        self.visitedCorners[corner] = False
def getStartState(self):
    return (self.startingPosition, self.visitedCorners)

def isGoalState(self, state):
    currState, currVis = state
    for corner in self.corners:
        if currVis[corner] is False:
            return False

    return True
```

Task 2 was provided by the course teacher. The task required us to solve the Corners problem, that is we have to make sure the dots in every corner of the maze are eaten.

Task 3

```
def cornersHeuristic(state, problem):
            ... ... ... ... ...
    visitedCorner = state[1]
    h = [0]
    for corner in corners:
        if visitedCorner[corner] is False:
            xy1= state[0]
            xy2 = corner
            distance =
util.manhattanDistance(xy1,xy2)
            h.append(distance)
    return max(h)
```

Here we take a list called h and initialize it to zero. Then we find the Manhattan distance from the pacman current position to every other not visited corner. As the heuristic, the maximum of the distances is returned. The explanation behind taking the maximum of the distances is as follows. The heuristic basically gives us the estimated cost to reach the goal state. But in our case we have no defined goal state. The corner that is farthest from the current position will obviously be visited at last and for this reason this corner is regarded as the goal states here. Once the farthest corner is visited, all the other corners are also visited and thus the goal state is reached. Thus when we return the maximum distance, we are returning the cost to reach the goal state eventually.

Task 4

```
def foodHeuristic(state, problem):
  position, foodGrid = state
  foodGrid = foodGrid.asList()
   h = [0]
   for food in foodGrid:
       distance = mazeDistance(position, food,
problem.startingGameState)
       h.append(distance)
   return max(h)
```

Here at first we take the food grid and a list h, initialized to 0. Then for every food in the food grid we calculate the maze distance from the current position of Pacman and store it in h. Finally, we return the maximum of the maze distance as the heuristic.
The reason for taking the maximum is the same as Task 3.

Task 5

```
def findPathToClosestDot(self, gameState):

        closest_heuristics = lambda position, foodGrid:
    min([util.manhattanDistance(position, food)
         for food in foodGrid.asList()])

        fringe = util.PriorityQueue()
        visited = []
        fringe.push((problem.getStartState(), []), 0)
        while True:
            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]),
                        closest_heuristics(startPosition,
food),
                    )
                    visited.append(currState)
        return None
```

Then in the *findPathToClosestDot()* function we write the main part of the code. The solution is again similar to UCS. The only difference here is that in the Priority Queue, only the heuristic (estimated) cost is used as priority. To calculate the heuristic we define a lambda

function called *closest_heuristics()* that calculates the Manhattan distance to every food from the current position and returns the minimum of it. The minimum is returned because we want the pacman to eat the closest food first. That is the main concept of greedy search. Thus the problem is solved.

—xxx—