Department of Computer Science and Engineering

Islamic University of Technology (IUT)

A subsidiary organ of OIC

CSE 4618: Artificial Intelligence

Lab Report 02

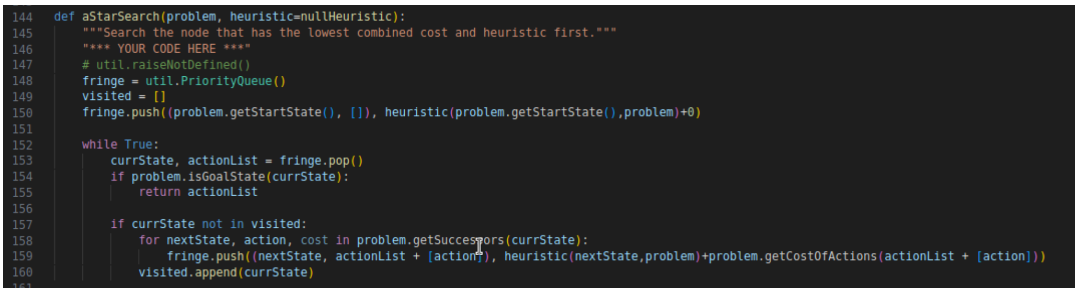
Name	**:**	**M M Nazmul Hossain**
Student ID	**:**	**200042118**
Semester	**:**	**6th**
Academic Year	**:**	**2022-2023**
Date of Submission	**:**	**27.02.2024**

Problem-1 (A Star)

Analysis

The first problem was to implement the Astar Search Algorithm. It would be provided the problem which gave us helpful functions like **getStartState** to start formulating the search function, the **getSuccessor** function to get the subsequent next Possible states based on all the possible actions for pacman, **getCostofActions** function to get the backward cost incurred by a particular list of actions, and the **isGoalState** function which checks whether the current state pacman is in is the goal state or not and a **heuristic** function which will provide a assumed forward cost from a particular node towards the goal node, which is preferably admissible and consistent, set to the **nullHeuristic(0)** by default.

Solution



```
144 def aStarSearch(problem, heuristic=nullHeuristic):
145     """Search the node that has the lowest combined cost and heuristic first."""
146     "*** YOUR CODE HERE ***"
147     # util.raiseNotDefined()
148     fringe = util.PriorityQueue()
149     visited = []
150     fringe.push((problem.getStartState(), []), heuristic(problem.getStartState(),problem)+0)
151 
152     while True:
153         currState, actionList = fringe.pop()
154         if problem.isGoalState(currState):
155             return actionList
156 
157         if currState not in visited:
158             for nextState, action, cost in problem.getSuccessors(currState):
159                 fringe.push((nextState, actionList + [action]), heuristic(nextState,problem)+problem.getCostOfActions(actionList + [action]))
160             visited.append(currState)
```

Explanation

The fringe was declared as a priority queue, where, the priority would be set according to the value of the following:

f = heuristicCost of a node + the backward cost from that node.

The rest of the algorithm is quite similar to the already implemented UCS algorithm which can be applied here with certain modifications.

This would mean that the algorithm would use both the backward cost (like in UCS) and an assumed potential cost till the goal state from the current node which will be provided by the heuristic function.

The backward cost will be calculated by following all the costs incurred by the path of actions taken to reach that state. The heuristic function will be provided by the user. By default it is set to the nullHeuristic which always sets the heuristic cost to 0. Surprisingly, this is both admissible and consistent. This way, the Astar algorithm would function exactly like the UCS algorithm, because without the heuristic cost, it will only take the backward cost into consideration where f = 0 + backward cost.

When using any heuristic function, it is preferable that the heuristic function be admissible when it comes to a tree search algorithm, which is the heuristic value provided by the function has to be greater than equal to 0 and less than or equal to the actual value to the goalstate. What also has to be taken into consideration while determining the heuristic function is that the calculations done to determine the heuristic cost shouldn't exceed the actual process, otherwise it becomes redundant in its goal to make the search process more efficient. When using the heuristic function in a graph search, it also has to be ensured that the provided heuristic cost is also consistent to maintain optimality of the search Algorithm.

A few common heuristic functions for the provided pacman problem could be considered. The Manhattan distance between the current state and the goal state could be used. This relaxes the problem since it doesn't consider the walls and any other requirements. Hence, the cost can never exceed the actual cost, so it remains both admissible and consistent.

Challenges

I had trouble reading the structure of the algorithm. I didn't realize the heuristic would be provided by the user, and I was trying to directly implement the heuristic function inside the Astar algorithm. Later I realized that it is a parameter provided by the user and it easy from there.

Problem-2 (Finding all the Corners)

Analysis

In the second problem, it required the definition of an abstract state representation of the Corners Problem, where there are four dots at each corner. Now, the problem had to be defined which included the follow functions:

1. Define part of the States of the problem
2. getStartState
3. isGoalState
4. getSuccessors

Solution

```
    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.visitedCorner = {}
        for corner in self.corners:
            self.visitedCorner[corner] = False


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

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

        return True
```

```
    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 ***"
            currPosition, currVisitedCorner = state
            x, y = currPosition
            dx, dy = Actions.directionToVector(action)
            nextX, nextY = int(x + dx), int(y + dy)
            hitsWall = self.walls[nextX][nextY]

            if not hitsWall:
                nextPosition = (nextX, nextY)
                nextVisitedCorner = deepcopy(currVisitedCorner)
                if nextPosition in self.corners:
                    nextVisitedCorner[nextPosition] = True

                nextState = (nextPosition, nextVisitedCorner)
                nextNode = (nextState, action, 1)
                successors.append(nextNode)


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

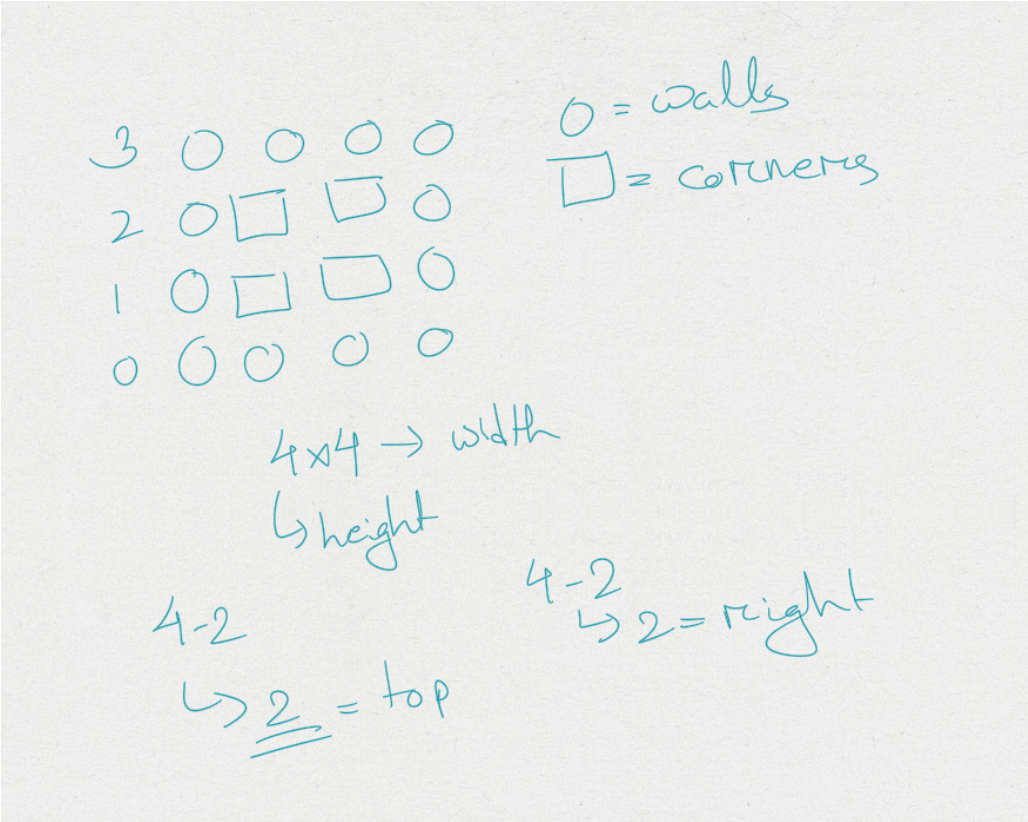
Explanation

The solution to this problem was provided by Bakhtiar Sir. At first, a new addition to the problem was the dictionary, or the visited corners array which was used to track which of the corners had already been visited by pacman in the problem.

The problem only considered the things which would be required for the corners problem and not redundant things like the position of the ghost, timer, pellets etc. It considered

the position of pacman, and its possible actions, which would be up, down, left and right provided there aren't any walls blocking its path. There are walls in the map at different positions and surrounding the entire map.

The corners that need to be visited are at the index (1,1), (1,top), (right,1) and (right,top).



Because of the surrounding walls, and the 0-based indexing, the value of top is height-2 and value of right is width-2.

The startState of the problem would be the starting position of pacman, and the visitedCorners array which says none of the corners have been visited.

The isGoalState function would check the visitedCorners array and if it finds any of the corners have not been visited yet, then it would return False. Else it would return true.

The getSuccessors function is tricky. The possible movements for pacman is up, down, left and right. However, if a wall was in the way of pacman's movements, then the successor function would not return that as a possible successor. The directionToVector function is used

for assistance in defining the movement of pacman. In the cases where pac man doesn't hit a wall, it is checked whether the position pacman is in is a corner or not. If it is, the visitedCorners array is updated. Now, Interestingly, python uses pass by reference in order to save an array. So, if the visitedCornersArray is updated, it will be updated globally which isn't desirable. The vistiedCorners array will be different for different nodes in the graph. For this reason, a deepcopy is made of the visitedCornersArray and the copy is then updated and stored alongside the state and is appended as a successor.

The getCostofActions was also predefined.

This problem only considers the position of pacman, the position of the walls and whether pacman has visited all four (4) corners or not. Pacman has 4 possible moves, up, down, left or right and a move becomes invalid if that move results in hitting a wall. So, the successor function only returns states for valid moves.

Challenges

Since the solution to this problem was provided by sir, the only challenge I faced was understanding the solution and how everything worked together.

Problem-3 (Corner Heuristics)

Analysis

The problem required us to design a heuristic function that would be both admissible and consistent for the corners problem. We had to define the foodHeuristic function and had previously implemented task 1 to solve this problem.

Solution

```
374 def cornersHeuristic(state, problem):
375     """
376     A heuristic for the CornersProblem that you defined.
377
378       state:   The current search state
379                (a data structure you chose in your search problem)
380
381       problem: The CornersProblem instance for this layout.
382
383     This function should always return a number that is a lower bound on the
384     shortest path from the state to a goal of the problem; i.e.  it should be
385     admissible (as well as consistent).
386     """
387     corners = problem.corners # These are the corner coordinates
388     walls = problem.walls # These are the walls of the maze, as a Grid (game.py)
389
390     "*** YOUR CODE HERE ***"
391     # return 0 # Default to trivial solution
392     currPosition, currVisitedCorners = state
393     maxDistance = 0
394     for corner in corners:
395         if currVisitedCorners[corner]==False:
396             distance = util.manhattanDistance(currPosition, corner)
397             maxDistance = max(maxDistance, distance)
398     return maxDistance
399
```

Explanation

To determine the appropriate heuristic functions, a few approaches were followed. The problem was relaxed by not considering any of the walls and only using the manhattanDistance between points.

At first, the distance between the current node and all the non-visited corners was calculated as the Heuristic value. However, that resulted in inconsistent heuristics.

After that, the idea of calculating the distance between the current position to the nearest non visited corner and from that to the next came to mind. However, it would be difficult to ensure that the solution would be admissible and consistent.

The next approach was to simply determine the manhattanDistance to the farthest non-visited corner and set the distance between that point and pacmans current position as the heuristic value.

This approach succeeded and got 3/3 in autograder.py. The interesting part in this approach was whether the heuristic value returned at the goal state was 0 or not didn't have to be checked since, when all the other corners are visited, pacman is on the final corner, the maxDistance between pacman and the final corner is set to 0 by the way the function is defined. So, the whether the count of non visited corners were 0 or not did not have to be checked. During the heuristic function definition.

Challenges

Before implementing the function using the farthest distance, I had tried to implement using the nearest corner, ie the min Distance. However, this faced the issue of non-zero heuristic value at the goal state. I didn't know how to return 0 when the goal state was reached in the heuristic function. However, before I dwelled too much on this, the idea of using the farthest corner came to mind and that inherently solved this challenge I was facing.

Problem-4 (Eating all the Dots)

Analysis

The problem required us to design a heuristic function that would be both admissible and consistent to allow pacman to eat all the food dots in an optimal way (fewest steps possible). We had to define the foodHeuristic function and had previously implemented task 1 to solve this problem.

Solution

```
462  def foodHeuristic(state, problem):
489      """
490      position, foodGrid = state
491      "*** YOUR CODE HERE ***"
492      # return 0
493      foodPositions = foodGrid.asList()
494      maxDistance = 0
495      minDistance = float('inf')
496      farthestFood = None
497      nearestFood = None
498      # print(foodPositions)
499      if not foodPositions:
500          return 0
501
502      for food in foodPositions:
503          distance = mazeDistance(position, food,problem.startingGameState)
504          if distance > maxDistance:
505              maxDistance = distance
506              farthestFood = food
507          if distance < minDistance:
508              minDistance = distance
509              nearestFood = food
510
511      return mazeDistance(position,nearestFood,problem.startingGameState) + mazeDistance(nearestFood, farthestFood,problem.startingGameState)
```

Explanation

The rewarding factor was limiting the number of nodes expanded in this problem. The trickySearch problem was used for testing.

A few different approaches were followed when designing the solution to this problem. First, The problem was relaxed by not considering the walls. The sequence in which the food had to be eaten was also not considered at first. Using the manhattanDistance between two points, the heuristic values were determined. Then, from the initial state, the food grid was traversed to determine the position of the food that was closest to Pacman. Returning the distance between

the nearest value as the heuristic expanded over 13000 nodes and took more than 7 seconds for precalculation which is only 2/4 in autograder.py.

Considering the food that was farthest away from Pacman, resulted in a little less than 10,000 nodes being expanded and only 3.9 seconds being required which is unfortunately only 3/4 in autograder.py.

After that, the sum of the manhattanDistance from the current position to the nearest food and the the distance from that food to the farthest away food was considered as the heuristic value. Using this, 4/4 on autograder.py was finally achieved. This algorithm expanded to less than 9000 nodes and took around 3.9 seconds. However, some bonus tasks had to be accomplished.

After searching the file, the mazeDistance algorithm was found, which, using bfs, determined the exact distance between two points, considering the walls inside the provided problem. Instead of using the manhattanDistance, the mazeDistance function could be used to determine a better heuristic. So, all the manhattanDistance was replaced with the new mazeDistance algorithm, which solved trickySearch by expanding only 1844 nodes and under 9.9 seconds. It resulted in 5/4 in autograder.py. It is also admissible and consistent because we're only considering the distance between the nearest and the farthest food points. It also isn't trivial because the exact cost isn't being calculated all the time.

Pacman would have to generally eat multiple food dots before all the dots are eaten, so, the cost would generally be higher.

When only two dots are remaining, this algorithm would return the distance between Pacman and the nearest dot, and the distance between the nearest and farthest node which is the exact remaining distance to be traversed as the heuristic value.

When there is only one dot remaining, the nearest and farthest node would be the same and the distance between them would be 0. So only the distance between Pacman and the remaining node is returned as the heuristic value.

When no food nodes are remaining, the heuristic value is returned as 0 via the if condition.

Challenges

Even though time was supposed to be an issue that should be taken under consideration, I wanted to design an algorithm that could solve this problem in a shorter period. I noticed that mazeDistance was a costly algorithm, so instead of using mazeDistance inside the loop, I wanted to use the manhattanDistance inside the loop to determine a possible nearby node and a possible farthest away node. Then use the mazeDistance formula as previously to return a heuristic value. Using this process, trickySearch required expanding over 2600 nodes however it was solved in under 2.4 seconds which is a significant improvement. Unfortunately, after running autograder.py, it showed that this heuristic was not consistent though I cannot explain why it isn't. I just quickly hurried back to my previous implementation which is both admissible and consistent and meets all the requirements that had been provided.

The main challenge I faced while solving this problem was thinking about how to get the bonus marks as I had found the manhattanDistance solution generally quickly. After searching through the file, I found the mazeDistance function, using which ultimately I solved the bonus problem as well.

Problem-5 (Suboptimal Search)

Analysis

The fifth problem was to implement a Food Problem which has a missing Goal State function, and the findPathToClosestDot and through that be able to design an algorithm that greedily eats the closest dot.

Solution

```
571     def isGoalState(self, state):
572         """
573         The state is Pacman's position. Fill this in with a goal test that will
574         complete the problem definition.
575         """
576         x,y = state
577 
578         "*** YOUR CODE HERE ***"
579         # util.raiseNotDefined()
580         if self.food[x][y]:
581             return True
582         return False
```

```
530     def findPathToClosestDot(self, gameState):
531         """
532         Returns a path (a list of actions) to the closest dot, starting from
533         gameState.
534         """
535         # Here are some useful elements of the startState
536         startPosition = gameState.getPacmanPosition()
537         food = gameState.getFood()
538         walls = gameState.getWalls()
539         problem = AnyFoodSearchProblem(gameState)
540 
541         "*** YOUR CODE HERE ***"
542         # util.raiseNotDefined()
543         return search.bfs(problem)
```

Explanation

The goal state was declared as reached if the position of the Pacman matched the position of any food in the food grid. It doesn't have to be the closest or furthest or follow any requirement. It can be any food on the list since our goal is to reach a suboptimal but fast solution.

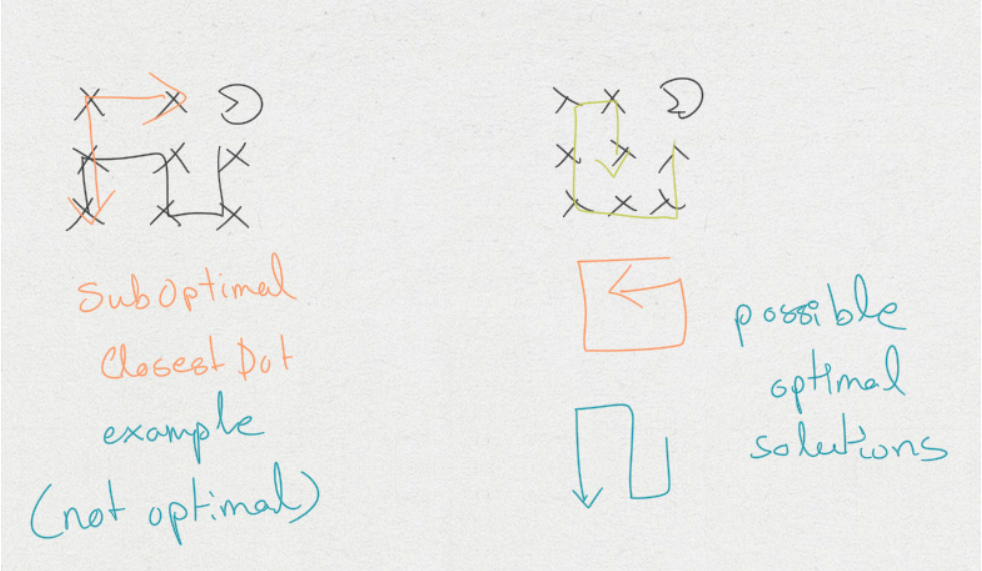
The path to the closest dot can be found using the bfs, ucs, or astar algorithm. Since the problem we defined always has a cost of 1, bfs would be able to reach the goal state fast and

efficiently which is why it was chosen. Since our goal is to greedily eat the closest dots, using bfs, we would be able to go down level first to each state and find the closest dot simply.

The goal of this problem was to reach a suboptimal but fast solution to the food search problem. From the previously implemented optimal food search algorithm, it can be seen that even for a medium-sized food search problem, it requires a very long time to precalculate all the routes because the graph to determine the optimal solution becomes too large.

However, if we try to obtain a more relaxed suboptimal solution, which is finding the path to the closest dot, then we can cut down on the time required for pre-processing. This ensures that if there is any food on the grid, it will be eventually eaten, even though not in the most optimal manner. Since the problem we are trying to solve is that all the dots have to be eaten, the optimal solution would be to eat all the dots most efficiently.

However, with the closestDotAgent, it just chooses whichever dot is the closest to it. As a result, it can sometimes leave certain dots in its path, and then they have to come back to eat them later. So, the solution generated by this agent isn't always optimal.



However, even though this solution is suboptimal, it is complete, and with this approach, even the bigMaze can be solved in a relatively shorter amount of time, as the time required to solve it using the optimal algorithm is immensely high.

Challenges

The main challenge I faced, while working on this problem, is designing the goal state. It was very difficult for me to understand what the goal state should be, since every time I am checking whether a node is a goal node or not, I cannot run a for loop across the food grid and determine the closest food node which would significantly slow down the process. Later, I realized that the goal node should be just to check whether or not pacman is in the same grid as a food grid. The nearest dot would be found using the algorithm.