HW1 Project Details Document

1. ***Question 1*** Implement the depth-first search (DFS) algorithm:

   - **Code Details:**

```
def depthFirstSearch(problem):
    """
    Search the deepest nodes in the search tree first.

    Your search algorithm needs to return a list of actions that reaches the
    goal. Make sure to implement a graph search algorithm.

    To get started, you might want to try some of these simple commands to
    understand the search problem that is being passed in:

    print "Start:", problem.getStartState()
    print "Is the start a goal?", problem.isGoalState(problem.getStartState())
    print "Start's successors:", problem.getSuccessors(problem.getStartState())
    """
    "*** YOUR CODE HERE ***"

    sourceNode = (problem.getStartState(), None, 0)
    nodesVisited = []
    backTrackDict = {problem.getStartState() : None}
    fringeList = util.Stack()
    fringeList.push(sourceNode)
    action = []
    closed = []

    while fringeList.isEmpty() != True:

            currentNode = fringeList.pop()
            nodesVisited.append(currentNode[0])

            if problem.isGoalState(currentNode[0]):
                goalState = currentNode
                break

            successorNodes = problem.getSuccessors(currentNode[0])
            for next in successorNodes:
                if next[0] not in nodesVisited:
                    backTrackDict[next[0]] = currentNode

                    """if problem.isGoalState(next[0]):
                        goalState = next
                        goalStateFound = True
                        break"""

                    fringeList.push(next)

    while backTrackDict[goalState[0]] != None:
        if goalState[1] != None:
            action.append(goalState[1])
        goalState = backTrackDict[goalState[0]]

    return list(reversed(action))
```

     - Initialize the source node with triplet values (startState, action(null), cost)
     - Initialize nodeVisited list to make the nodes which are visited
     - Initialize backtrack list which will contain parent node of the current node
     - Take a stack for putting successors of a node
     - Put the source node in the stack
     - Initialize action list which will contain the valid actions to reach from start to goal
     - Initialize closed list which will mark the nodes which are explored
     - Loop until the stack is empty:
       - Remove the top node from stack
       - Mark it as visited

        - If the current node is goal then assign it in goal state and break from the loop
        - Else continue and get the successor nodes of the current node
        - For each successor nodes:
            - If node is not already visited then put the successor node in the backtracking list with current node as its parent
    - Loop over backtracking list until we reach the start node:
        - If action of the goal state is not null (source node) then append the action in the action list
        - Set the goal state to the next element of the backtracking list and continue finding all the valid actions

    - Return the reversed action list so that it stores the actions from startNode to the goalNode.

- **Output:**

Python Command	Output
python pacman.py -l tinyMaze -p SearchAgent --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l tinyMaze -p SearchAgent --frameTime 0 [SearchAgent] using function depthFirstSearch [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 10 in 0.0 seconds Search nodes expanded: 15 Pacman emerges victorious! Score: 500 Average Score: 500.0 Scores: 500.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$
python pacman.py -l mediumMaze -p SearchAgent --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l mediumMaze -p SearchAgent --frameTime 0 [SearchAgent] using function depthFirstSearch [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 130 in 0.0 seconds Search nodes expanded: 146 Pacman emerges victorious! Score: 380 Average Score: 380.0 Scores: 380.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$
python pacman.py -l bigMaze -z .5 -p SearchAgent --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l bigMaze -z .5 -p SearchAgent --frameTime 0 [SearchAgent] using function depthFirstSearch [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 210 in 0.0 seconds Search nodes expanded: 390 Pacman emerges victorious! Score: 300 Average Score: 300.0 Scores: 300.0 Win Rate: 1/1 (1.00) Record: Win

	Gourabs-MacBook-Pro:search gourabbhattacharyya$

2. **Question 2** Implement the breadth-first search (BFS) algorithm:

- **Code Details:**

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"

    sourceNode = (problem.getStartState(), None, 0)
    nodesVisited = [problem.getStartState()]
    closed = []
    backTrackDict = {problem.getStartState(): None}
    fringeList = util.Queue()
    fringeList.push(sourceNode)
    action = []

    while fringeList.isEmpty() != True:
            currentNode = fringeList.pop()

            if problem.isGoalState(currentNode[0]):
                goalState = currentNode
                break


            if currentNode[0] not in closed:
                successorNodes = problem.getSuccessors(currentNode[0])
                closed.append(currentNode[0])

                for next in successorNodes:
                    if next[0] not in nodesVisited:
                        backTrackDict[next[0]] = currentNode

                        """if problem.isGoalState(next[0]):
                            goalState = next
                            goalStateFound = True
                            break"""

                        fringeList.push(next)
                        nodesVisited.append(next[0])



    while backTrackDict[goalState[0]] != None:
        if goalState[1] != None:
            action.append(goalState[1])
        goalState = backTrackDict[goalState[0]]

    return list(reversed(action))
    #util.raiseNotDefined()
```

- Initialize the source node with triplet values (startState, action(null), cost)
- Initialize nodeVisited list to make the nodes which are visited
- Initialize closed list which will mark the nodes which are explored
- Initialize backtrack list which will contain parent node of the current node
- Take a queue for putting successors of a node
- Push the source node in the queue as first element
- Initialize action list which will contain the valid actions to reach from start to goal

- Loop until the stack is empty:
  - Pop the first element of the queue as a current node
  - If current state is goal state then assign it as goalNode and break from the loop
  - If current node is not in explored list then get all the successor nodes of the current node
  - Append the current node in the explored list
  - For each successor nodes:
    - If it's not visited then add it in the backtracking list with parent as current node
    - Push the node in the queue
    - Mark the node as visited
- Loop over backtracking list until we reach the start node:
  - If action of the goal state is not null (source node) then append the action in the action list
  - Set the goal state to the next element of the backtracking list and continue finding all the valid actions
- Return the reversed action list so that it stores the actions from startNode to the goalNode.

- **Output:**

Python Command	Output
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs --frameTime 0 [SearchAgent] using function bfs [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 68 in 0.0 seconds Search nodes expanded: 269 Pacman emerges victorious! Score: 442 Average Score: 442.0 Scores: 442.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime 0 [SearchAgent] using function bfs [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 210 in 0.0 seconds Search nodes expanded: 620 Pacman emerges victorious! Score: 300 Average Score: 300.0 Scores: 300.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$

3. **Question 3** Implement the uniform-cost search (UCS) algorithm

- **Code Details:**

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"

    sourceNode = (problem.getStartState(), None, 0)
    nodesVisited = []
    backTrackDict = {problem.getStartState(): None}
    fringePList = util.PriorityQueue()
    fringePList.push(sourceNode, sourceNode[2])
    nodeCost = {problem.getStartState() : sourceNode[2]}
    action = []


    while fringePList.isEmpty() != True:
            currentNode = fringePList.pop()
            nodesVisited.append(currentNode[0])

            if problem.isGoalState(currentNode[0]):
                goalState = currentNode
                break

            successorNodes = problem.getSuccessors(currentNode[0])
            for next in successorNodes:
                if next[0] not in nodesVisited:

                    totalCost = nodeCost[currentNode[0]] + next[2]

                    if next[0] in nodeCost:
                        if nodeCost[next[0]] < totalCost:
                            continue    #search for optimal path
                        else:
                            fringePList.update(next, totalCost)
                    else:
                        fringePList.push(next, totalCost)

                    backTrackDict[next[0]] = currentNode
                    nodeCost[next[0]] = totalCost

                    """if problem.isGoalState(next[0]):
                        goalState = next
                        goalStateFound = True
                        break"""

    while backTrackDict[goalState[0]] != None:
        if goalState[1] != None:
            action.append(goalState[1])
        goalState = backTrackDict[goalState[0]]

    return list(reversed(action))
```

- Initialize the source node with triplet values (startState, action(null), cost)
- Initialize nodeVisited list to make the nodes which are visited
- Initialize backtrack list which will contain parent node of the current node
- Take a priority queue for putting successors of a node
- Push the source node in the queue as first element and set its priority as the path cost
- Initialize a nodeCost dictionary which will keep track of the nodes already visited and cost to reach that node from the source
- Initialize action list which will contain the valid actions to reach from start to goal
- Loop until the stack is empty:
    - Pop the element with least priority from the priority queue as current node
    - Mark that node as visited
    - If current node is goal then set it as goalNode and break from the loop

        - Explore all the successor of the current node
        - For each successor node if not marked as visited:
            - Calculate total cost = cost retrieved from nodeCost dict for the current node + cost of successor node
            - If successor node already present in the nodeCost dict and its costValue is less than new total cost then no need to take any action and we will continue to find any other optimal path
            - If new total cost is less then update the queue with the new cost for that node
            - If the node is not visited before then add that in the queue with the total cost as priority
            - Put that node in the backtracking list and set current node as its parent
            - Add the node and its cost in nodeCost dictionary.
    - Loop over backtracking list until we reach the start node:
        - If action of the goal state is not null (source node) then append the action in the action list
        - Set the goal state to the next element of the backtracking list and continue finding all the valid actions
    - Return the reversed action list so that it stores the actions from startNode to the goalNode.

- **Output:**

Python Command	Output
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs --frameTime 0 [SearchAgent] using function ucs [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 68 in 0.0 seconds Search nodes expanded: 275 Pacman emerges victorious! Score: 442 Average Score: 442.0 Scores: 442.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$
python pacman.py -l mediumDottedMaze -p StayEastSearchAgent --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l mediumDottedMaze -p StayEastSearchAgent --frameTime 0 Path found with total cost of 1 in 0.0 seconds Search nodes expanded: 186 Pacman emerges victorious! Score: 646 Average Score: 646.0 Scores: 646.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$
python pacman.py -l mediumScaryMaze -p StayWestSearchAgent --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l mediumScaryMaze -p StayWestSearchAgent --frameTime 0

	Path found with total cost of 68719479864 in 0.0 seconds Search nodes expanded: 108 Pacman emerges victorious! Score: 418 Average Score: 418.0 Scores: 418.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$

4. **Question 4** Implement A* algorithm

- **Code Details:**

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"

    sourceNode = (problem.getStartState(), None, 0)
    backTrackDict = {problem.getStartState(): None}

    fVal = {}
    fVal[sourceNode] = heuristic(problem.getStartState(), problem)

    fringeAList = util.PriorityQueue()
    fringeAList.push(sourceNode, fVal[sourceNode])

    nodesVisited = []
    nodeCost = {problem.getStartState(): sourceNode[2]}
    closed = []
    action = []


    while fringeAList.isEmpty() != True:
            currentNode = fringeAList.pop()
            nodesVisited.append(currentNode[0])

            if problem.isGoalState(currentNode[0]):
                goalState = currentNode
                break

            successorNodes = problem.getSuccessors(currentNode[0])
            for next in successorNodes:
                if next[0] not in nodesVisited:

                    totalCost = nodeCost[currentNode[0]] + next[2]
                    fVal[next] = totalCost + heuristic(next[0], problem)

                    if next[0] in nodeCost:
                        if nodeCost[next[0]] < totalCost:
                            continue
                        else:
                            fringeAList.update(next, fVal[next])
                    else:
                        fringeAList.push(next, fVal[next])

                    backTrackDict[next[0]] = currentNode
                    nodeCost[next[0]] = totalCost


    while backTrackDict[goalState[0]] != None:
        if goalState[1] != None:
            action.append(goalState[1])
        goalState = backTrackDict[goalState[0]]

    return list(reversed(action))
```

- Initialize the source node with triplet values (startState, action(null), cost)
- Initialize backtrack list which will contain parent node of the current node
- Initialize a fVal dictionary to hold the value of total path cost[f(n) = g(n) + h(n)]and put the heuristic value of the first node
- Take a priority queue for putting successors of a node
- Push the source node in the queue as first element and set its priority as the total path cost
- Initialize nodeVisited list to make the nodes which are visited
- Initialize a nodeCost dictionary which will keep track of the nodes already visited and cost to reach that node from the source
- Initialize closed list which will maintain the already explored list
- Initialize action list which will contain the valid actions to reach from start to goal
- Loop until the stack is empty:
    - Pop the element with least priority from the priority queue as current node
    - Mark that node as visited
    - If current node is goal then set it as goalNode and break from the loop
    - Explore all the successor of the current node
    - For each successor node if not marked as visited:
        - Calculate total cost = cost retrieved from nodeCost dict for the current node + cost of successor node
        - Calculate fValue(f(n)) = total cost(g(n)) + heuristic(h(n)) of the successor node
        - If successor node already present in the nodeCost dict and its costValue is less than new total cost then no need to take any action and we will continue to find any other optimal path
        - If new total cost is less then update the queue with the new cost for that node
        - If the node is not visited before then add that in the queue with the total cost as priority
        - Put that node in the backtracking list and set current node as its parent
        - Add the node and its cost in nodeCost dictionary.
- Loop over backtracking list until we reach the start node:
    - If action of the goal state is not null (source node) then append the action in the action list
    - Set the goal state to the next element of the backtracking list and continue finding all the valid actions
- Return the reversed action list so that it stores the actions from startNode to the goalNode.

- **Output:**

Python Command	Output
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic --frameTime 0 [SearchAgent] using function astar and heuristic manhattanHeuristic [SearchAgent] using problem type PositionSearchProblem Path found with total cost of 210 in 0.0 seconds Search nodes expanded: 549

	Pacman emerges victorious! Score: 300 Average Score: 300.0 Scores: 300.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$

5. **Question 5** Implement the CornersProblem search problem

- **Code Details:**

```
class CornersProblem(search.SearchProblem):
    """
    This search problem finds paths through all four corners of a layout.

    You must select a suitable state space and successor function
    """

    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.startState = (self.startingPosition, ())

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

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        return len(state[1]) == len(self.corners)
        #util.raiseNotDefined()
```

```
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 ***"
        nodesVisited = state[1]

        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        hitsWall = self.walls[nextx][nexty]

        if not hitsWall:
            nextNodePosition = (nextx, nexty)

            if nextNodePosition in self.corners:
                if nextNodePosition not in nodesVisited:
                    nodesVisited = nodesVisited + (nextNodePosition, )
            successors.append( ((nextNodePosition, nodesVisited), action, 1))

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

def getCostOfActions(self, actions):
    """
    Returns the cost of a particular sequence of actions.  If those actions
    include an illegal move, return 999999.  This is implemented for you.
    """
    if actions == None: return 999999
    x,y= self.startingPosition
    for action in actions:
        dx, dy = Actions.directionToVector(action)
        x, y = int(x + dx), int(y + dy)
        if self.walls[x][y]: return 999999
    return len(actions)
```

- Initialize the start state in __init__ method as tuple of starting position
- In getStartState() method return the start state
- Use isGoalState() method to check is the current state is goal state or not
- In getSuccessor() method:
    - For each valid direction:
        - Initialize node visited list with the current node
        - Store the current position in x, y values
        - Check if the next positions are hitting the wall or not
        - If not hitting the wall then get the nextNodePosition
        - If the nextNodePosition in the maze corners and is they are not marked as visited then append it in the nodesVisited list
        - Append the next node with next position along with valid direction and cost to the successor list.
    - Return the successor list

- **Output:**

Python Command	Output
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem --frameTime 0 [SearchAgent] using function bfs [SearchAgent] using problem type CornersProblem Path found with total cost of 28 in 0.0 seconds Search nodes expanded: 435 Pacman emerges victorious! Score: 512 Average Score: 512.0 Scores: 512.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem --frameTime 0 [SearchAgent] using function bfs [SearchAgent] using problem type CornersProblem Path found with total cost of 106 in 0.3 seconds Search nodes expanded: 2448 Pacman emerges victorious! Score: 434 Average Score: 434.0 Scores: 434.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$

6. **Question 6** Implement a heuristic for the CornersProblem in cornersHeuristic

- **Code Details:**

```
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 ***"

    sourceNode = state[0]
    nodesVisited = [state[1]]
    nodesNotVisited = []
    admiHeuristicValue = sys.maxint

    for corner in corners:
        if corner not in nodesVisited:
            nodesNotVisited.append(corner)

    for corner in nodesNotVisited:
        if(sourceNode != corner):
            minDist = util.manhattanDistance(sourceNode, corner)
            if (minDist < admiHeuristicValue):
                admiHeuristicValue = minDist
        else:
            admiHeuristicValue = 0

    return admiHeuristicValue # Default to trivial solution
```

- Get the corners and walls from the given problem
- Initialize the source node
- Initialize the nodesVisied and set current node
- Initialize nodesNotVisited list
- Initialize the heuristic value as the maximum value
- For each corners:
  - If it's not already marked as visited then append it in the nodesNotVisited list
- For each item in nodesNotVisited:
  - If item is not source then find the minimum distance which is manhattanDistance between source and current node
  - If the minimum distance is less than heuristic value then set this as new heuristic value
  - If item is source node then set heuristic as 0
- Return the heuristic value

- **Output:**

Python Command	Output
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5 --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5 --frameTime 0 Path found with total cost of 106 in 0.2 seconds Search nodes expanded: 2164 Pacman emerges victorious! Score: 434

	Average Score: 434.0 Scores: 434.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$

7. **Question 7** foodHeuristic with a consistent heuristic for the FoodSearchProblem

- **Code Details:**

```
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']
    """
    "*** YOUR CODE HERE ***"
    position, foodGrid = state

    allFoods = foodGrid.asList()
    heuristicVal = 0

    if len(allFoods) == 0:
        return 0
    else:
        for item in allFoods:
            if item not in problem.walls:
                minDist = mazeDistance(position, item, problem.startingGameState)
                if minDist > heuristicVal:
                    heuristicVal = minDist

    return heuristicVal
```

- Get the position and foodGrid from the game state
- Convert foodGrid as food list and assign in allFoods
- Initialize the heuristic as 0
- If there are no food left then return 0
- Otherwise for each food in allFoods:

- If the food is not on the walls then find the mazeDistance using current satte position, food and start state.
- If new distance is greater than heuristic then assign that as new heuristic value

➢ Return the calculated heuristic value

- **Output:**

Python Command	Output
python pacman.py -l trickySearch -p AStarFoodSearchAgent --frameTime 0	Gourabs-MacBook-Pro:search gourabbhattacharyya$ python pacman.py -l trickySearch -p AStarFoodSearchAgent --frameTime 0 Path found with total cost of 60 in 23.0 seconds Search nodes expanded: 4328 Pacman emerges victorious! Score: 570 Average Score: 570.0 Scores: 570.0 Win Rate: 1/1 (1.00) Record: Win Gourabs-MacBook-Pro:search gourabbhattacharyya$