Faculty of Engineering

Artificial Intelligence

AI assignment 2

Names:
Amr Bekhiet (50)
Radwa Elamsry (25)
Aliaa Othman (44)

November 25, 2017

1 Problem Statement

In this project, your Pacman agent will find paths through his maze world, both to reach a particular location and to collect food efficiently. You will build general search algorithms and apply them to Pacman scenarios.

2 Finding a Fixed Food Dot using Depth First Search

2.1 Code

```
"""
"*** YOUR CODE HERE ***"
""" initialize frontier list with the start state """
frontier = util.Stack()
""" initialize state for the search problem """
state = {}
state["node"] = problem.getStartState()
state["path"] = []
frontier.push(state)
""" initialize the explored set """
explored = set()
"""loop on frontier list """
while not frontier.isEmpty():
    """ get the node with smallest cost """
    temp_state = frontier.pop()
    current_state = temp_state["node"]
    current_path = temp_state["path"]

    """ if it is the goal just return the path """
    if (problem.isGoalState(current_state)):
        return current_path
    """ if the explored set doesn't have the current_state  just add it """
    if (not current_state in explored):
        explored.add(current_state)
    """ for each successor of the current state """
    for neighbour in problem.getSuccessors(current_state):
        if not neighbour[0] in explored:
            result_node = {}
            result_node["node"] = neighbour[0]
            result_node["path"] = current_path + [neighbour[1]]
            frontier.push(result_node)

return current_path
```

2.2 Algorithm

1. initialize the frontier list as stack
2. initialize the game state consisting of : node and path
3. push the first state into frontier
4. initialize the explored list
5. loop on frontier list until it is empty :
    (a) pop from the frontier list
    (b) check if the popped state is the goal state return the path
    (c) add the popped state in the explored set
    (d) for each successor of the current state popped from the frontier
        i. if the neighbour isn't in the explored list :
        ii. initialize a new state with this neighbour and put the path of it as the current path plus the action to this neighbour.
        iii. push the neighbour state in the frontier list.
6. return the current path of the current node

3 Breadth First Search

3.1 Code

```
frontier = util.Queue()
""" initialize state for the search problem """
state = {}
state["node"] = problem.getStartState()
state["path"] = []
frontier.push(state)
""" initialize the explored set """
explored = set()
"""loop on frontier list """
while not frontier.isEmpty():
    """ get the node with smallest cost """
    temp_state = frontier.pop()
    current_state = temp_state["node"]
    current_path = temp_state["path"]

    """ if it is the goal just return the path """
    if (problem.isGoalState(current_state)):
        "print current_path"
        return current_path
    """ if the explored set doesn't have the current_state  just add it """
    if (not current_state in explored):
        explored.add(current_state)
    """ for each successor of the current state """
    for neighbour in problem.getSuccessors(current_state):
        if neighbour[0] not in explored:
            isInFrontier = False
            for node in frontier.list:  # not in frontier
                if neighbour[0] == node["node"]:
                    isInFrontier = True
            if not isInFrontier:
                result_node = {}
                result_node["node"] = neighbour[0]
                result_node["path"] = current_path + [neighbour[1]]
                frontier.push(result_node)

return current_path
```

3.2 Algorithm

1. initialize the frontier list as Queue.
2. initialize the game state consisting of : node and path
3. push the first state into frontier

4. initialize the explored list
5. loop on frontier list until it is empty :
    (a) pop from the frontier list
    (b) check if the popped state is the goal state return the path
    (c) add the popped state in the explored set
    (d) for each successor of the current state popped from the frontier
        i. if the neighbour isn't in the explored list and not in the frontier :
        ii. initialize a new state with this neighbour and put the path of it as the current path plus the action to this neighbour.
        iii. push the neighbour state in the frontier list.
6. return the current path of the current node

4 Varying the Cost Function (UCS)

4.1 Code

```
def uniformCostSearch(problem):

    return aStarSearch(problem)
```

4.2 Algorithm

Same algorithm as A* but with null heuristic.

5 A* Search

5.1 Code

```
def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    if problem is None:
        raise ValueError("Search Problem cannot be None")

    priorityQueue = util.PriorityQueue()
    initState = problem.getStartState()

    if initState is None:
        raise ValueError("Initial State cannot be None")

    # Initialization
    isExplored = set()
    inQueue = set()
    currCost = {}
    # For each state (pos + direction) store action to achieve this state
    actions = {initState: (initState, None)}
    # Heuristic cost to reach goal from init state = 0 + g(n)
    initCost = heuristic(initState, problem)
    # Prioritize the state with accumlated cost on total path cost = (g(n)+f(n))
    priorityQueue.push(initState, initCost)
    inQueue.add(initState)
    currCost[initState] = 0

    # A* search logic
    while not priorityQueue.isEmpty():
        currState = priorityQueue.pop()
        isExplored.add(currState)

        # Goal is achieved
        if problem.isGoalState(currState):
            return getActions(currState, actions)

        # For each (nextstate, action, cost)
        for (nextState, action, cost) in problem.getSuccessors(currState):
            # If not explored and not currently in queue to be explored
            if not ((nextState in inQueue) or (nextState in isExplored)):
                # Add candidate node to the priority queue with cost = g(n) + f(n)
                priorityQueue.push(nextState, currCost[currState] + cost + heuristic(nextState, problem))
                inQueue.add(nextState)
                # Set cost to reach successor = cost to reach parent + cost to reach successor from parent
                currCost[nextState] = currCost[currState] + cost
                # Store actions taken from parent to reach successor
                actions[nextState] = (currState, action)
            # Else if successor is visited and explored yet
            elif nextState in inQueue:
                # Update successor (cost & parent & action taken) only incase reached with less cost from another parent
                if (currCost[currState] + cost) < currCost[nextState]:
                    priorityQueue.update(nextState, currCost[currState] + cost + heuristic(nextState, problem))
                    actions[nextState] = (currState, action)
    return []
```

5.2 Algorithm

1. initialize a Priority Queue.
2. initialize a Dictionary of key = node, value = cost reached sofar

3. initialize a Dictionary of key = node, value = Tuple (Parent, Action to reach child from parent)
4. initialize a Set of explored nodes
5. initialize a Set of nodes currently in Queue
6. push the first state into Priority Queue
7. initialize the explored Set
8. initialize the InQueue Set
9. loop on Priority Queue until it is empty :
   (a) pop from the frontier list
   (b) add the popped state in the explored set
   (c) check if the popped state is the goal state return the path recursively generated
   (d) for each successor of the current state popped from the Priority Queue
      i. if the neighbour isn't in the explored list And not in Queue:
      ii. initialize a new state with this neighbour
      iii. add neighbour state to InQueue Set
      iv. else if not explored push the neighbour state in the priority Queue to update its priority.
10. return the current path of the current node

6 Finding All the Corners

6.1 Code

```
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"
        # know the corner corresponding bit in the mask
        self.cornersMaskPos = {self.corners[0]: 0, self.corners[1]: 1, self.corners[2]: 2, self.corners[3]: 3}
        # our goal to visit all the corners so the mask has all ones
        self.GOAL = (2 ** 4) - 1
    "check if the position is one of the corners and if it is set the corresponding bit in mask"
    def getStateMask(self,mask, position):
        if position in self.cornersMaskPos:
            mask = (mask | (1 << self.cornersMaskPos[position]))
        return mask
```

```
    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "state consisted of position of packman and the mask showing the visited corners"
        state = (self.startingPosition, self.getStateMask(0, self.startingPosition))
        return state

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        return (state[1] == self.GOAL)
```

```python
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 = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        if not self.walls[nextx][nexty]:
            position = (nextx, nexty)
            # check if the position is one of the corners and if it is set it in the mask.
            successors.append(((position, self.getStateMask(state[1], position)), action, 1))

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

6.2 Algorithm

- a Mask is used to track the visited corners.
- cornersMaskPos is a dictionary to map from one corner to a bit in the mask.
- problem goal is to visit all the corners so when the mask is all ones we reach the goal.
- state of the problem consisted of : position and the mask showing the visited corners.
- In getting successor for each position we checked if the position is one of the corners and if it is set the corresponding bit.

7 Corners Problem: Heuristic

7.1 Code

```
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)
    maskPos = problem.cornersMaskPos # relate each corner to a bit in mask

    "*** YOUR CODE HERE ***"
    # mask is an indication of the visited corner
    position, mask = state
    points = []
    # get the unvisited corners
    for corner in corners:
        if (mask & (1 << maskPos[corner])) == 0:
            points = points + [corner]
    return getCornersHeuristic(position, points) # Default to trivial solution

def getNextPoint(position, points):
    minDistance = 999999
    nearestPoint = position
    # for each corner in the unvisited corners , get the distance from the current position to it , then get the minDistance
    for point in points:
        distance = util.manhattanDistance(position, point)
        if distance <= minDistance:
            minDistance = distance
            nearestPoint = point
    return (nearestPoint, minDistance)

def getCornersHeuristic(position, points):
    if len(points) == 0:
        return 0
    # position and minDistance is changed
    nearestPoint, minDistance = getNextPoint(position, points)
    # the point is visited now so remove it
    points.remove(nearestPoint)
    return minDistance + getCornersHeuristic(nearestPoint, points)
```

7.2 Algorithm

- First we get the unvisited corners.

- each time we calculate the heuristic by calculating the distances between the position and unvisited corners , then choose the minimum distance.
- choose the new position as the position of the visited corner from the previous step , then go to step 2 again until all corners are visited.
- return the cumulative distance as the heuristic.

8 Eating All The Dots

8.1 Code

```
position, foodGrid = state
"*** YOUR CODE HERE ***"
distances = [0]
for food in foodGrid.asList():
    # use the mazeDistance funtion to explore less nodes than the manhattan does
    distances.append(mazeDistance(position, food, problem.startingGameState))
# return the furthest distance
return max(distances)
```

8.2 Algorithm

- Get the distance between packman position and each food in the grid.
- choose the farthest distance to be the heuristic.
- getting the distance by using mazeDistance explores less number of nodes than the manhattan distance explores.

9 Suboptimal Search

9.1 Code

```
def isGoalState(self, state):
    """
    The state is Pacman's position. Fill this in with a goal test that will
    complete the problem definition.
    """
    x,y = state
    # when we find any food we reach the goal
    return self.food[x][y]
```

```
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 ***"
    return search.breadthFirstSearch(problem)
```

9.2 Algorithm

- The goal state is reached when we find any food .
- We choose to run the problem with bfs to find the shortest path to any food.

10 Auto Grader

```
Finished at 0:49:34

Provisional grades
==================
Question q1: 3/3
Question q2: 3/3
Question q3: 3/3
Question q4: 3/3
Question q5: 3/3
Question q6: 3/3
Question q7: 5/4
Question q8: 3/3
------------------
Total: 26/25
```