Project1 实验报告

search

github链接:

Question 1

本题要求实现深度优先搜索。深度优先搜索可通过递归实现，可借助一个栈来实现非递归的深度优先搜索。当需要进入到某个节点的子节点就把当前节点压入栈保存。又因为题目要求返回动作的列表，可以在每次进入到下一个节点的时候，把进入到这个节点的动作也压入栈中，最后在返回动作列表的时候，对这个栈进行一下处理换成列表即可。

注意到，对于每个节点，expand函数只能调用一次，在调用expand函数的时候，就算是对这个节点进行了遍历，而节点的孩子信息需要调用expand函数才能获得，而深度优先搜索有时候需要回溯，回溯到先前的节点的时候就不能再次调用expand函数获取孩子信息了。为解决这个问题，使用一个字典记录节点的孩子信息，这样回溯的时候就能够获知其孩子信息了。

代码及注释如下:

```
def depthFirstSearch(problem):
    frontier = util.Stack() #用栈存节点
    pathSt = util.Stack() #存路径
    expanded = [] #记录已经扩展过的节点
    children_dict = {} #记录节点及其对应孩子列表
    frontier.push(problem.getStartState()) #把起点放进栈中
    while not frontier.isEmpty(): #当栈不空
        node = frontier.pop() #取出栈顶元素
        if problem.isGoalState(node): #判断是否目标状态，若是则返回路径
            path = []
            while not pathSt.isEmpty():
                path.insert(0,pathSt.pop()) #返回的是列表，需要处理一下栈中元素
            return path
        if node not in expanded: #检查节点是否扩展过，如果没扩展
            children = problem.expand(node)  #调用expand函数获取其孩子
            children_dict[node] = children #记录其孩子信息
        else:
            children = children_dict[node] #如果已经调用过expand，则从字典中取孩子信息
        if node not in expanded:
            expanded.append(node)

        canExpand = 0 #一个标志，检查当前节点是否还能向深处扩展
        for nextState, action, cost in children: #对节点的每一个孩子
            if nextState not in expanded: #看看孩子是否有未扩展过的节点，若有
                frontier.push(node) #把当前节点入栈
                node = nextState #移动到这个孩子节点
                frontier.push(node) #这个孩子节点也入栈
                canExpand = 1 #修改标志
                pathSt.push(action) #加入动作
                break #跳出循环
        if not canExpand: #如果这个节点不能再向深处扩展
            pathSt.pop() #动作栈中弹出栈顶元素
    return []
```

Question 2

本题要求实现广度优先搜索，广度优先搜索可借助一个先进先出队列来进行实现。但是由于util里面写好的Queue类没有很方便的检查一个元素是否在Queue中的接口，因此，这里用了一个list来记录进入队列的节点。由于广度优先搜索是按层遍历的，遍历节点的顺序并不是最终路径的顺序。在广度优先搜索中，扩展一个节点的时候，其父节点是确定的。因此，借助了两个字典，分别记录进入节点的父节点以及进入节点的动作，返回结果时可以方便的找到动作顺序。

代码及注释如下：

```
def breadthFirstSearch(problem):
    expanded = [] #记录扩展过的节点
    recorded = [] #记录被添加到先进先出队列的节点
    stateFatherDic = {} #记录状态及其父节点，用于返回结果时得到动作路径
    stateActionDic = {} #记录状态及其对应的动作，用于返回结果时得到动作路径
    frontier = util.Queue() #先进先出的队列，按这个队列的顺序进行遍历
    frontier.push(problem.getStartState()) #将起点放到队列中
    recorded.append(problem.getStartState())
    while not frontier.isEmpty(): #当队列不空的时候
        node = frontier.pop() #取出队头元素
        if problem.isGoalState(node): #如果是目标状态，往回走得到动作路径
            path = []
            curNode = node
            while curNode in stateFatherDic:
                path.insert(0,stateActionDic[curNode])
                curNode = stateFatherDic[curNode]
            return path
        if node not in expanded: #如果节点没有扩展过
            expanded.append(node) #将节点加到扩展过的列表中
            children = problem.expand(node) #调用expand函数获取其孩子信息
            for nextState, action, cost in children: #对其每一个孩子
                if nextState not in recorded: #如果这个孩子没有被加入队列
                    stateFatherDic[nextState] = node #记录信息
                    stateActionDic[nextState] = action
                    recorded.append(nextState)
                    frontier.push(nextState) #将这个孩子加入队列
    return []
```

Question 3

本题要求实现A*搜索算法。A*搜索算法接受一个启发式函数作为输入，它通过如下函数来计算每个节点的优先级

$$f(n) = g(n) + h(n)$$

其中f(n)是综合优先级，g(n)是节点n距离起点的代价，h(n)是节点n距离终点的预计代价，是由启发式函数确定的。在每次运算过程中，每次都从优先队列中选取综合优先级最小的节点作为下一个要遍历的节点。借助一个PriorityQueue来实现此算法。

代码及注释如下：

```
def aStarSearch(problem, heuristic=nullHeuristic):
    nodeFatherDic = {} #记录进入节点的父节点
    nodeActionDic = {} #记录进入节点的动作
```

```
    #this cost not include cost from heuristic
    nodeMinCost = {} #节点距离起点的最小代价
    openSet = util.PriorityQueue() #优先队列，可选的遍历节点

    closeSet = [] #遍历过的节点
    openSet.push(problem.getStartState(),0) #将起点放入优先队列，并设代价为0
    nodeMinCost[problem.getStartState()] = 0
    while not openSet.isEmpty(): #当优先队列不空的时候
        node = openSet.pop() #取出代价最小的节点
        closeSet.append(node) #加入遍历过的节点集
        if problem.isGoalState(node): #检查是否目标状态，若是则处理一下动作序列并返回
            path = []
            curNode = node
            while curNode in nodeFatherDic:
                path.insert(0,nodeActionDic[curNode])
                curNode = nodeFatherDic[curNode]
            return path

        children = problem.expand(node) #调用expand函数获取其孩子信息
        for nextState, action, cost in children: #对其每一个孩子
            if nextState not in closeSet: #如果没有遍历过
                pastCost = nodeMinCost[node] + cost #计算此节点经其父节点到起点的代价
                totalCost = nodeMinCost[node] + cost +
heuristic(nextState,problem) #计算总和代价
                if nextState not in nodeMinCost or nodeMinCost[nextState] >
pastCost: #如果距离起点代价更小，则代价信息
                    nodeMinCost[nextState] = pastCost
                    nodeFatherDic[nextState] = node #更新动作信息
                    nodeActionDic[nextState] = action
                    openSet.update(nextState, totalCost) #更新优先队列中的节点代价信
息

    return []
```

Question 4

本题要求补充完成CornersProblem类，实现getStartState, isGoalState, expand, getNextState函数。在这个问题中，目标状态是搜索完四个角落。因此，除了当前位置以外，状态中需要记录搜索过的哪些角落以及角落的位置信息。

角落有四个，因此可以使用一个由0，1组成的四元组表示每一个角落是否被遍历过。

在getStartState函数中，返回的是当前位置，以及角落位置和对应的角落是否有被遍历过的四元组。

在isGoalState函数中，首先获取当前位置，然后检查当前位置是否在其中一个角落，若是，读取角落位置信息以及对应角落是否有被遍历过的四元组，因为元组不可更改，所以复制一份转成列表，修改列表中当前位于的角落为遍历过的状态。检查这个列表，看看是否四个角落都遍历过了，若是，则返回True，否则返回False

在getNextState函数中，输入当前状态和动作，返回下一个状态。首先获取当前位置、角落位置、遍历角落的情况。在这个函数中借助了Actions.directionToVector函数，再加上当前位置，能够得到下一个位置。检查下一个位置是否角落，如果是角落，修改遍历角落信息。将（下一个位置，（角落位置，角落遍历信息））作为下一个状态返回。

在expand函数中，需要扩展此节点，返回其孩子信息。借助getNextState函数，可以得到从当前位置出发，采取所有可能动作之后下一个所有可能状态。把这些所有可能状态加到children列表里面，最后作为结果返回即可。

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

    You must select a suitable state space and child 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.reachedCorners = (0,0,0,0)

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        if self.startingPosition in self.reachedCorners:
            self.reachedCorners[self.startingPosition] = 1
        return (self.startingPosition, (self.corners,self.reachedCorners))

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
        "*** YOUR CODE HERE ***"
        curPos = state[0]
        current_reached_corners = list(state[1][1])
        corner_pos_list = list(self.corners)
        if curPos in self.corners:
            idx = corner_pos_list.index(curPos)
            current_reached_corners[idx] = 1
        for item in current_reached_corners:
            if item == 0:
                return False
        return True

    def expand(self, state):
        """
        Returns child 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, (child,
            action, stepCost), where 'child' is a child to the current
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that child
        """

        children = []
        for action in self.getActions(state):
            # Add a child state to the child list if the action is legal
            # You should call getActions, getActionCost, and getNextState.
            "*** YOUR CODE HERE ***"
            next_state = self.getNextState(state,action)
            children.append((next_state,action,1))

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

    def getActions(self, state):
        possible_directions = [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]
        valid_actions_from_state = []
        for action in possible_directions:
            x, y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                valid_actions_from_state.append(action)
        return valid_actions_from_state

    def getActionCost(self, state, action, next_state):
        assert next_state == self.getNextState(state, action), (
            "Invalid next state passed to getActionCost().")
        return 1

    def getNextState(self, state, action):
        assert action in self.getActions(state), (
            "Invalid action passed to getActionCost().")
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        "*** YOUR CODE HERE ***"
        current_reached_corners = list(state[1][1])
        if (nextx, nexty) in self.corners:
            idx = self.corners.index((nextx,nexty))
            current_reached_corners[idx] = 1
        return ((nextx, nexty), (self.corners,tuple(current_reached_corners)))

    def getCostOfActionSequence(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)
```

Question 5

本题要求实现一个启发式函数，针对遍历四个角落的问题来设计。由于启发式函数估计的是节点到终点的代价，而本题的目标状态是需要遍历完四个节点。启发式函数在估计代价的时候除了要知道当前位置以外，还需要知道当前状态下哪些角落已经遍历过了，哪些还没有，以及四个角落的位置信息。

当前状态如果是目标状态，则返回0，否则，进行以下操作来估计遍历剩下角落的代价。首先，看看当前哪些角落没有遍历过，把没有遍历过的角落加到一个列表里面。用一个变量total_cost记录启发式函数要返回的代价，对于没有遍历过的那些角落，选取距离当前位置最近的那个，然后假设将当前位置移动到那里，接下来在剩下的没遍历过的角落里面，选取距离移动后位置最近的那个，以此类推，计算这样遍历完没遍历过的角落的代价总和，作为total_cost返回。

```
def cornersHeuristic(state, problem):
    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 ***"
    current_position = state[0]
    if(problem.isGoalState(state)):
        return 0
    current_reached_corners = state[1][1]

    not_visited_corners = []
    for i in range(4):
        if current_reached_corners[i] == 0:
            not_visited_corners.append(corners[i])

    total_cost = 0
    while len(not_visited_corners): #当列表中还有没遍历的节点的时候，进入循环
        to_one_corner_cost = 99999 #一次移动的代价，用于比较，以选出最小代价的下一个角落点
        for c in not_visited_corners: #选出距离“当前”最近的节点
            if abs(c[0] - current_position[0]) + abs(c[1] - current_position[1])
< to_one_corner_cost:
                to_one_corner_cost = abs(c[0] - current_position[0]) + abs(c[1]
- current_position[1])
                next_corner = c
        current_position = next_corner #将“当前”节点移动到选出的节点
        total_cost += to_one_corner_cost #记录这次移动的代价
        not_visited_corners.remove(current_position) #把这个选出的节点从未遍历节点列表
中拿走
    return total_cost #返回移动代价总和
```

Question 6

本题要求设计一个在尽可能少的步数中吃掉所有豆子的启发式算法。首先计算当前位置距离最远豆子的曼哈顿距离，然后统计有多少豆子不在当前位置去往最远豆子的方向上，两者加起来作为启发式函数的结果返回。

```
def foodHeuristic(state, problem):
    position, foodGrid = state
    "*** YOUR CODE HERE ***"
    foodGridList = foodGrid.asList()
    farthest_food = position #将曼哈顿距离最远的食物位置初始化为当前位置
    farthest_distance = 0  #最远曼哈顿距离代价初始化为0
    for food in foodGridList: #对于每一个豆子
        if abs(position[0] - food[0]) + abs(position[1] - food[1]) >
farthest_distance: #如果有曼哈顿距离更大的豆子，则更新
            farthest_food = food
            farthest_distance = abs(position[0] - food[0]) + abs(position[1] -
food[1])
    diff = position[0] - farthest_food[0] #用当前位置的横坐标减去最远豆子的横坐标
    count = 0 #计数器，查看有多少豆子不在当前位置去往最远豆子的方向上
    for food in foodGridList: #对于每一个豆子
        if diff > 0 and position[0] < food[0]: #如果最远豆子在当前位置的左边，并且这个
豆子在当前位置的右边，计数器加一
            count += 1
        elif diff < 0 and position[0] > food[0]: #如果最远豆子在当前位置的右边，并且这
个豆子在当前位置的左边，计数器加一
            count += 1
        elif diff == 0 and position[0] != food[0]: #如果最远豆子和当前位置横坐标相等，
并且这个豆子和它们横坐标不等，计数器加一
            count += 1
    return farthest_distance + count #返回最远豆子的距离加上计数器的值
```

Question 7

本题要求补充ClosestDotSearchAgent类的实现，以找到距离最近点的一条路径。主要为补全findPathToClosestDot函数，返回距离最近点的动作列表。首先，初始化一个返回动作的空列表。接下来，获取豆子位置的列表，for循环遍历检查哪一个豆子是距离最近的豆子，把problem的goal设为这个最近豆子的位置。最后，调用之前实现过的A*搜索函数，将曼哈顿距离作为启发式函数，获得动作的列表，作为结果返回即可。

```
class ClosestDotSearchAgent(SearchAgent):
    "Search for all food using a sequence of searches"
    def registerInitialState(self, state):
        self.actions = []
        currentState = state
        while(currentState.getFood().count() > 0):
            nextPathSegment = self.findPathToClosestDot(currentState) # The
missing piece
            self.actions += nextPathSegment
            for action in nextPathSegment:
                legal = currentState.getLegalActions()
                if action not in legal:
                    t = (str(action), str(currentState))
                    raise Exception('findPathToClosestDot returned an illegal
move: %s!\n%s' % t)
                currentState = currentState.generateChild(0, action)
        self.actionIndex = 0
        print('Path found with cost %d.' % len(self.actions))

    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 ***"
        path = []
        foodGridList = food.asList()
        closest = foodGridList.pop() #取出一个豆子的位置，初始化最近的豆子位置变量
        for f in foodGridList: #遍历每一个豆子，寻找最近的一个
            if abs(f[0] - startPosition[0]) + abs(f[1] - startPosition[1]) <
abs(closest[0] - startPosition[0]) + abs(closest[1] - startPosition[1]):
                closest = f
        problem.goal = closest #把目标设置为最近的豆子位置
        path = search.aStarSearch(problem, manhattanHeuristic) #调用搜索函数进行搜索
        return path
```

附录

题目完成得分

```
(2021ai) D:\code\artificial_intelligence\Project1>python autograder.py -q q1
Starting on 10-11 at 19:51:11

Question q1
===========
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:               ['0:A->B', '0:B->D', '0:D->G']
***     expanded_states:        ['A', 'B', 'D']
*** PASS: test_cases\q1\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_manypaths.test
***     solution:               ['0:A->B1', '0:B1->C', '0:C->D', '0:D->E1', '0:E1->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'D', 'E1', 'F']
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 246
***     nodes expanded:         269

### Question q1: 4/4 ###


Finished at 19:51:11

Provisional grades
==================
Question q1: 4/4
------------------
Total: 4/4
```

```
(2021ai) D:\code\artificial_intelligence\Project1>python autograder.py -q q2
Starting on 10-11 at 19:51:27

Question q2
===========
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 4/4 ###


Finished at 19:51:27

Provisional grades
==================
Question q2: 4/4
------------------
Total: 4/4
```

```
(2021ai) D:\code\artificial_intelligence\Project1>python autograder.py -q q3
Starting on 10-11 at 19:51:44

Question q3
===========
*** PASS: test_cases\q3\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q3\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q3\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q3: 4/4 ###
```

```
(2021ai) D:\code\artificial_intelligence\Project1>python autograder.py -q q4
Note: due to dependencies, the following tests will be run: q2 q4
Starting on 10-11 at 19:52:01

Question q2
===========
*** PASS: test_cases\q2\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q2\graph_bfs_vs_dfs.test
***     solution:               ['1:A->G']
***     expanded_states:        ['A', 'B']
*** PASS: test_cases\q2\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q2\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']
*** PASS: test_cases\q2\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 4/4 ###


Question q4
===========
*** PASS: test_cases\q4\corner_tiny_corner.test
***     pacman layout:          tinyCorner
***     solution length:                28

### Question q4: 3/3 ###


Finished at 19:52:01

Provisional grades
==================
Question q2: 4/4
Question q4: 3/3
------------------
Total: 7/7
```

```
(2021ai) D:\code\artificial intelligence\Project1>python autograder.py -q q5
Note: due to dependencies, the following tests will be run: q3 q5
Starting on 10-11 at 19:52:22

Question q3
===========
*** PASS: test_cases\q3\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q3\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q3\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q3: 4/4 ###


Question q5
===========
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'West', 'West', 'South', 'South', 'East', 'East',
  'East', 'East', 'South', 'South', 'South', 'South', 'South', 'South', 'West', 'West', 'South', 'South', 'South', 'West', 'West', 'East', 'East', 'North', 'North', 'North', 'East', 'East', 'East', 'East', 'East', 'East', 'East', 'East'
, 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'East', 'East', 'Nort
h', 'North', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'North', 'North', 'North', 'North', 'North', 'North', 'North', 'West', 'West', 'North', 'North', 'East', 'East', 'North', 'North']
path length: 106
*** PASS: Heuristic resulted in expansion of 602 nodes

### Question q5: 3/3 ###


Finished at 19:52:22

Provisional grades
==================
Question q3: 4/4
Question q5: 3/3
------------------
Total: 7/7
```

```
(2021ai) D:\code\artificial intelligence\Project1>python autograder.py -q q6
Note: due to dependencies, the following tests will be run: q3 q6
Starting on 10-11 at 19:52:50

Question q3
===========
*** PASS: test_cases\q3\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q3\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q3\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q3\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q3\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q3\graph_manypaths.test
***     solution:               ['1:A->C', '0:C->D', '1:D->F', '0:F->G']
***     expanded_states:        ['A', 'B1', 'C', 'B2', 'D', 'E1', 'F', 'E2']

### Question q3: 4/4 ###


Question q6
===========
*** PASS: test_cases\q6\food_heuristic_1.test
*** PASS: test_cases\q6\food_heuristic_10.test
*** PASS: test_cases\q6\food_heuristic_11.test
*** PASS: test_cases\q6\food_heuristic_12.test
*** PASS: test_cases\q6\food_heuristic_13.test
*** PASS: test_cases\q6\food_heuristic_14.test
*** PASS: test_cases\q6\food_heuristic_15.test
*** PASS: test_cases\q6\food_heuristic_16.test
*** PASS: test_cases\q6\food_heuristic_17.test
*** PASS: test_cases\q6\food_heuristic_2.test
*** PASS: test_cases\q6\food_heuristic_3.test
*** PASS: test_cases\q6\food_heuristic_4.test
*** PASS: test_cases\q6\food_heuristic_5.test
*** PASS: test_cases\q6\food_heuristic_6.test
*** PASS: test_cases\q6\food_heuristic_7.test
*** PASS: test_cases\q6\food_heuristic_8.test
*** PASS: test_cases\q6\food_heuristic_9.test
*** FAIL: test_cases\q6\food_heuristic_grade_tricky.test
***     expanded nodes: 8957
***     thresholds: [15000, 12000, 9000, 7000]
```

```
### Question q6: 4/4 ###


Finished at 19:52:56

Provisional grades
==================
Question q3: 4/4
Question q6: 4/4
------------------
Total: 8/8
```

```
(2021ai) D:\code\artificial_intelligence\Project1>python autograder.py -q q7
Starting on 10-11 at 19:53:48

Question q7
===========
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_1.test
***     pacman layout:          Test 1
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_10.test
***     pacman layout:          Test 10
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_11.test
***     pacman layout:          Test 11
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_12.test
***     pacman layout:          Test 12
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_13.test
***     pacman layout:          Test 13
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_2.test
***     pacman layout:          Test 2
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_3.test
***     pacman layout:          Test 3
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_4.test
***     pacman layout:          Test 4
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_5.test
***     pacman layout:          Test 5
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
```

```
*** PASS: test_cases\q7\closest_dot_6.test
***     pacman layout:          Test 6
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_7.test
***     pacman layout:          Test 7
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_8.test
***     pacman layout:          Test 8
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q7\closest_dot_9.test
***     pacman layout:          Test 9
***     solution length:                1

### Question q7: 3/3 ###


Finished at 19:53:48

Provisional grades
==================
Question q7: 3/3
------------------
Total: 3/3
```

```
Finished at 19:50:19

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

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

完整代码文件

search.py

```
# search.py
# ---------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
```

```
#
# Attribution Information: The Pacman AI projects were developed at UC Berkeley.
# The core projects and autograders were primarily created by John DeNero
# (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# Student side autograding was added by Brad Miller, Nick Hay, and
# Pieter Abbeel (pabbeel@cs.berkeley.edu).


"""
In search.py, you will implement generic search algorithms which are called by
Pacman agents (in searchAgents.py).
"""

import util

class SearchProblem:
    """
    This class outlines the structure of a search problem, but doesn't implement
    any of the methods (in object-oriented terminology: an abstract class).

    You do not need to change anything in this class, ever.
    """

    def getStartState(self):
        """
        Returns the start state for the search problem.
        """
        util.raiseNotDefined()

    def isGoalState(self, state):
        """
          state: Search state

        Returns True if and only if the state is a valid goal state.
        """
        util.raiseNotDefined()

    def expand(self, state):
        """
          state: Search state

        For a given state, this should return a list of triples, (child,
        action, stepCost), where 'child' is a child to the current
        state, 'action' is the action required to get there, and 'stepCost' is
        the incremental cost of expanding to that child.
        """
        util.raiseNotDefined()

    def getActions(self, state):
        """
          state: Search state

        For a given state, this should return a list of possible actions.
        """
        util.raiseNotDefined()

    def getActionCost(self, state, action, next_state):
        """
```

```
          state: Search state
          action: action taken at state.
          next_state: next Search state after taking action.

        For a given state, this should return the cost of the (s, a, s')
transition.
        """
        util.raiseNotDefined()

    def getNextState(self, state, action):
        """
          state: Search state
          action: action taken at state

        For a given state, this should return the next state after taking action
from state.
        """
        util.raiseNotDefined()

    def getCostOfActionSequence(self, actions):
        """
         actions: A list of actions to take

        This method returns the total cost of a particular sequence of actions.
        The sequence must be composed of legal moves.
        """
        util.raiseNotDefined()


def tinyMazeSearch(problem):
    """
    Returns a sequence of moves that solves tinyMaze.  For any other maze, the
    sequence of moves will be incorrect, so only use this for tinyMaze.
    """
    from game import Directions
    s = Directions.SOUTH
    w = Directions.WEST
    return  [s, s, w, s, w, w, s, w]


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()))
    """
    "*** YOUR CODE HERE ***"
    frontier = util.Stack()
    pathSt = util.Stack()
    expanded = []
    children_dict = {}
```

```
    frontier.push(problem.getStartState())
    while not frontier.isEmpty():
        node = frontier.pop()
        if problem.isGoalState(node):
            path = []
            while not pathSt.isEmpty():
                path.insert(0,pathSt.pop())
            return path
        if node not in expanded:
            children = problem.expand(node)
            children_dict[node] = children
        else:
            children = children_dict[node]
        if node not in expanded:
            expanded.append(node)

        canExpand = 0
        for nextState, action, cost in children:
            if nextState not in expanded:
                frontier.push(node)
                node = nextState
                frontier.push(node)
                canExpand = 1
                pathSt.push(action)
                break
        if not canExpand:
            pathSt.pop()
    return []

def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    expanded = []
    recorded = []
    stateFatherDic = {}
    stateActionDic = {}
    frontier = util.Queue()
    frontier.push(problem.getStartState())
    recorded.append(problem.getStartState())
    while not frontier.isEmpty():
        node = frontier.pop()
        if problem.isGoalState(node):
            path = []
            curNode = node
            while curNode in stateFatherDic:
                path.insert(0,stateActionDic[curNode])
                curNode = stateFatherDic[curNode]
            return path
        if node not in expanded:
            expanded.append(node)
            children = problem.expand(node)
            for nextState, action, cost in children:
                if nextState not in recorded:
                    stateFatherDic[nextState] = node
                    stateActionDic[nextState] = action
                    recorded.append(nextState)
                    frontier.push(nextState)
    return []
```

```
def nullHeuristic(state, problem=None):
    """
    A heuristic function estimates the cost from the current state to the
nearest
    goal in the provided SearchProblem.  This heuristic is trivial.
    """
    return 0

def aStarSearch(problem, heuristic=nullHeuristic):
    """Search the node that has the lowest combined cost and heuristic first."""
    "*** YOUR CODE HERE ***"
    nodeFatherDic = {}
    nodeActionDic = {}
    #this cost not include cost from heuristic
    nodeMinCost = {}
    openSet = util.PriorityQueue()

    closeSet = []
    openSet.push(problem.getStartState(),0)
    nodeMinCost[problem.getStartState()] = 0
    while not openSet.isEmpty():
        node = openSet.pop()
        closeSet.append(node)
        #print("current node",node)
        if problem.isGoalState(node):
            #print("Goal")
            path = []
            curNode = node
            while curNode in nodeFatherDic:
                path.insert(0,nodeActionDic[curNode])
                curNode = nodeFatherDic[curNode]
            return path

        children = problem.expand(node)
        for nextState, action, cost in children:
            if nextState not in closeSet:
                pastCost = nodeMinCost[node] + cost
                totalCost = nodeMinCost[node] + cost +
heuristic(nextState,problem)
                #print("nextState:",nextState,"cost:",totalCost)
                if nextState not in nodeMinCost or nodeMinCost[nextState] >
pastCost:
                    #print("update nextState info, nextState:",nextState)
                    nodeMinCost[nextState] = pastCost
                    nodeFatherDic[nextState] = node
                    nodeActionDic[nextState] = action
                    openSet.update(nextState, totalCost)

    return []


# Abbreviations
bfs = breadthFirstSearch
dfs = depthFirstSearch
astar = aStarSearch
```

searchAgent.py

```
# searchAgents.py
# ---------------
# Licensing Information:  You are free to use or extend these projects for
# educational purposes provided that (1) you do not distribute or publish
# solutions, (2) you retain this notice, and (3) you provide clear
# attribution to UC Berkeley, including a link to http://ai.berkeley.edu.
#
# Attribution Information: The Pacman AI projects were developed at UC Berkeley.
# The core projects and autograders were primarily created by John DeNero
# (denero@cs.berkeley.edu) and Dan Klein (klein@cs.berkeley.edu).
# Student side autograding was added by Brad Miller, Nick Hay, and
# Pieter Abbeel (pabbeel@cs.berkeley.edu).


"""
This file contains all of the agents that can be selected to control Pacman.  To
select an agent, use the '-p' option when running pacman.py.  Arguments can be
passed to your agent using '-a'.  For example, to load a SearchAgent that uses
depth first search (dfs), run the following command:

> python pacman.py -p SearchAgent -a fn=depthFirstSearch

Commands to invoke other search strategies can be found in the project
description.

Please only change the parts of the file you are asked to.  Look for the lines
that say

"*** YOUR CODE HERE ***"

The parts you fill in start about 3/4 of the way down.  Follow the project
description for details.

Good luck and happy searching!
"""

from game import Directions
from game import Agent
from game import Actions
import util
import time
import search
import math

class GoWestAgent(Agent):
    "An agent that goes West until it can't."

    def getAction(self, state):
        "The agent receives a GameState (defined in pacman.py)."
        if Directions.WEST in state.getLegalPacmanActions():
            return Directions.WEST
        else:
            return Directions.STOP
```

```
#####################################################
# This portion is written for you, but will only work #
#       after you fill in parts of search.py          #
#####################################################

class SearchAgent(Agent):
    """
    This very general search agent finds a path using a supplied search
    algorithm for a supplied search problem, then returns actions to follow that
    path.

    As a default, this agent runs DFS on a PositionSearchProblem to find
    location (1,1)

    Options for fn include:
      depthFirstSearch or dfs
      breadthFirstSearch or bfs


    Note: You should NOT change any code in SearchAgent
    """

    def __init__(self, fn='depthFirstSearch', prob='PositionSearchProblem',
heuristic='nullHeuristic'):
        # Warning: some advanced Python magic is employed below to find the
right functions and problems

        # Get the search function from the name and heuristic
        if fn not in dir(search):
            raise AttributeError(fn + ' is not a search function in search.py.')
        func = getattr(search, fn)
        if 'heuristic' not in func.__code__.co_varnames:
            print('[SearchAgent] using function ' + fn)
            self.searchFunction = func
        else:
            if heuristic in globals().keys():
                heur = globals()[heuristic]
            elif heuristic in dir(search):
                heur = getattr(search, heuristic)
            else:
                raise AttributeError(heuristic + ' is not a function in
searchAgents.py or search.py.')
            print('[SearchAgent] using function %s and heuristic %s' % (fn,
heuristic))
            # Note: this bit of Python trickery combines the search algorithm
and the heuristic
            self.searchFunction = lambda x: func(x, heuristic=heur)

        # Get the search problem type from the name
        if prob not in globals().keys() or not prob.endswith('Problem'):
            raise AttributeError(prob + ' is not a search problem type in
SearchAgents.py.')
        self.searchType = globals()[prob]
        print('[SearchAgent] using problem type ' + prob)

    def registerInitialState(self, state):
        """
```

```
        This is the first time that the agent sees the layout of the game
        board. Here, we choose a path to the goal. In this phase, the agent
        should compute the path to the goal and store it in a local variable.
        All of the work is done in this method!

        state: a GameState object (pacman.py)
        """
        if self.searchFunction == None: raise Exception("No search function
provided for SearchAgent")
        starttime = time.time()
        problem = self.searchType(state) # Makes a new search problem
        self.actions  = self.searchFunction(problem) # Find a path
        totalCost = problem.getCostOfActionSequence(self.actions)
        print('Path found with total cost of %d in %.1f seconds' % (totalCost,
time.time() - starttime))
        if '_expanded' in dir(problem): print('Search nodes expanded: %d' %
problem._expanded)

    def getAction(self, state):
        """
        Returns the next action in the path chosen earlier (in
        registerInitialState).  Return Directions.STOP if there is no further
        action to take.

        state: a GameState object (pacman.py)
        """
        if 'actionIndex' not in dir(self): self.actionIndex = 0
        i = self.actionIndex
        self.actionIndex += 1
        if i < len(self.actions):
            return self.actions[i]
        else:
            return Directions.STOP

class PositionSearchProblem(search.SearchProblem):
    """
    A search problem defines the state space, start state, goal test, child
    function and cost function.  This search problem can be used to find paths
    to a particular point on the pacman board.

    The state space consists of (x,y) positions in a pacman game.

    Note: this search problem is fully specified; you should NOT change it.
    """

    def __init__(self, gameState, costFn = lambda x: 1, goal=(1,1), start=None,
warn=True, visualize=True):
        """
        Stores the start and goal.

        gameState: A GameState object (pacman.py)
        costFn: A function from a search state (tuple) to a non-negative number
        goal: A position in the gameState
        """
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        if start != None: self.startState = start
        self.goal = goal
```

```
        self.costFn = costFn
        self.visualize = visualize
        if warn and (gameState.getNumFood() != 1 or not
gameState.hasFood(*goal)):
            print('Warning: this does not look like a regular search maze')

        # For display purposes
        self._visited, self._visitedlist, self._expanded = {}, [], 0 # DO NOT
CHANGE

    def getStartState(self):
        return self.startState

    def isGoalState(self, state):
        isGoal = state == self.goal

        # For display purposes only
        if isGoal and self.visualize:
            self._visitedlist.append(state)
            import __main__
            if '_display' in dir(__main__):
                if 'drawExpandedCells' in dir(__main__._display):
#@UndefinedVariable
                    __main__._display.drawExpandedCells(self._visitedlist)
#@UndefinedVariable

        return isGoal

    def expand(self, state):
        """
        Returns child 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,
         (child, action, stepCost), where 'child' is a
         child to the current state, 'action' is the action
         required to get there, and 'stepCost' is the incremental
         cost of expanding to that child
        """

        children = []
        for action in self.getActions(state):
            nextState = self.getNextState(state, action)
            cost = self.getActionCost(state, action, nextState)
            children.append( ( nextState, action, cost) )

        # Bookkeeping for display purposes
        self._expanded += 1 # DO NOT CHANGE
        if state not in self._visited:
            self._visited[state] = True
            self._visitedlist.append(state)

        return children

    def getActions(self, state):
        possible_directions = [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]
        valid_actions_from_state = []
```

```
        for action in possible_directions:
            x, y = state
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                valid_actions_from_state.append(action)
        return valid_actions_from_state

    def getActionCost(self, state, action, next_state):
        assert next_state == self.getNextState(state, action), (
            "Invalid next state passed to getActionCost().")
        return self.costFn(next_state)

    def getNextState(self, state, action):
        assert action in self.getActions(state), (
            "Invalid action passed to getActionCost().")
        x, y = state
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        return (nextx, nexty)

    def getCostOfActionSequence(self, actions):
        """
        Returns the cost of a particular sequence of actions. If those actions
        include an illegal move, return 999999.
        """
        if actions == None: return 999999
        x,y= self.getStartState()
        cost = 0
        for action in actions:
            # Check figure out the next state and see whether its' legal
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]: return 999999
            cost += self.costFn((x,y))
        return cost

class StayEastSearchAgent(SearchAgent):
    """
    An agent for position search with a cost function that penalizes being in
    positions on the West side of the board.

    The cost function for stepping into a position (x,y) is 1/2^x.
    """
    def __init__(self):
        self.searchFunction = search.uniformCostSearch
        costFn = lambda pos: .5 ** pos[0]
        self.searchType = lambda state: PositionSearchProblem(state, costFn, (1,
1), None, False)

class StayWestSearchAgent(SearchAgent):
    """
    An agent for position search with a cost function that penalizes being in
    positions on the East side of the board.

    The cost function for stepping into a position (x,y) is 2^x.
    """
    def __init__(self):
```

```
        self.searchFunction = search.uniformCostSearch
        costFn = lambda pos: 2 ** pos[0]
        self.searchType = lambda state: PositionSearchProblem(state, costFn)

def manhattanHeuristic(position, problem, info={}):
    "The Manhattan distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = problem.goal
    return abs(xy1[0] - xy2[0]) + abs(xy1[1] - xy2[1])

def euclideanHeuristic(position, problem, info={}):
    "The Euclidean distance heuristic for a PositionSearchProblem"
    xy1 = position
    xy2 = problem.goal
    return ( (xy1[0] - xy2[0]) ** 2 + (xy1[1] - xy2[1]) ** 2 ) ** 0.5

#####################################################
# This portion is incomplete.  Time to write 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 child 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.reachedCorners = (0,0,0,0)

    def getStartState(self):
        """
        Returns the start state (in your state space, not the full Pacman state
        space)
        """
        "*** YOUR CODE HERE ***"
        if self.startingPosition in self.reachedCorners:
            self.reachedCorners[self.startingPosition] = 1
        return (self.startingPosition, (self.corners,self.reachedCorners))

    def isGoalState(self, state):
        """
        Returns whether this search state is a goal state of the problem.
        """
```

```
        "*** YOUR CODE HERE ***"
        curPos = state[0]
        #print("state[0]",state[0],"  state[1]:",state[1], "  state[1]
[1]:",state[1][1])
        current_reached_corners = list(state[1][1])
        corner_pos_list = list(self.corners)
        if curPos in self.corners:
            idx = corner_pos_list.index(curPos)
            current_reached_corners[idx] = 1
        for item in current_reached_corners:
            if item == 0:
                return False
        return True

    def expand(self, state):
        """
        Returns child 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, (child,
            action, stepCost), where 'child' is a child to the current
            state, 'action' is the action required to get there, and 'stepCost'
            is the incremental cost of expanding to that child
        """

        children = []
        for action in self.getActions(state):
            # Add a child state to the child list if the action is legal
            # You should call getActions, getActionCost, and getNextState.
            "*** YOUR CODE HERE ***"
            next_state = self.getNextState(state,action)
            children.append((next_state,action,1))

        curPos = state[0]
        corner_pos_list = list(self.corners)
        current_reached_corners = list(state[1][1])
        if curPos in self.corners:
            idx = corner_pos_list.index(curPos)
            current_reached_corners[idx] = 1
        self._expanded += 1 # DO NOT CHANGE
        return children

    def getActions(self, state):
        possible_directions = [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]
        valid_actions_from_state = []
        for action in possible_directions:
            x, y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                valid_actions_from_state.append(action)
        return valid_actions_from_state

    def getActionCost(self, state, action, next_state):
        assert next_state == self.getNextState(state, action), (
            "Invalid next state passed to getActionCost().")
        return 1
```

```
    def getNextState(self, state, action):
        assert action in self.getActions(state), (
            "Invalid action passed to getActionCost().")
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        "*** YOUR CODE HERE ***"
        current_reached_corners = list(state[1][1])
        if (nextx, nexty) in self.corners:
            idx = self.corners.index((nextx,nexty))
            current_reached_corners[idx] = 1
        return ((nextx, nexty), (self.corners,tuple(current_reached_corners)))

    def getCostOfActionSequence(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)


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 ***"
    current_position = state[0]
    if(problem.isGoalState(state)):
        return 0
    current_reached_corners = state[1][1]

    not_visited_corners = []
    for i in range(4):
        if current_reached_corners[i] == 0:
            not_visited_corners.append(corners[i])

    total_cost = 0
    while len(not_visited_corners):
        to_one_corner_cost = 99999
```

```
        for c in not_visited_corners:
            if abs(c[0] - current_position[0]) + abs(c[1] - current_position[1])
< to_one_corner_cost:
                to_one_corner_cost = abs(c[0] - current_position[0]) + abs(c[1]
- current_position[1])
                next_corner = c
        current_position = next_corner
        total_cost += to_one_corner_cost
        not_visited_corners.remove(current_position)
    return total_cost

class AStarCornersAgent(SearchAgent):
    "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
    def __init__(self):
        self.searchFunction = lambda prob: search.aStarSearch(prob,
cornersHeuristic)
        self.searchType = CornersProblem

class FoodSearchProblem:
    """
    A search problem associated with finding the a path that collects all of the
    food (dots) in a Pacman game.

    A search state in this problem is a tuple ( pacmanPosition, foodGrid ) where
      pacmanPosition: a tuple (x,y) of integers specifying Pacman's position
      foodGrid:       a Grid (see game.py) of either True or False, specifying
remaining food
    """
    def __init__(self, startingGameState):
        self.start = (startingGameState.getPacmanPosition(),
startingGameState.getFood())
        self.walls = startingGameState.getWalls()
        self.startingGameState = startingGameState
        self._expanded = 0 # DO NOT CHANGE
        self.heuristicInfo = {} # A dictionary for the heuristic to store
information

    def getStartState(self):
        return self.start

    def isGoalState(self, state):
        return state[1].count() == 0

    def expand(self, state):
        "Returns child states, the actions they require, and a cost of 1."
        children = []
        self._expanded += 1 # DO NOT CHANGE
        for action in self.getActions(state):
            next_state = self.getNextState(state, action)
            action_cost = self.getActionCost(state, action, next_state)
            children.append( ( next_state, action, action_cost) )
        return children

    def getActions(self, state):
        possible_directions = [Directions.NORTH, Directions.SOUTH,
Directions.EAST, Directions.WEST]
        valid_actions_from_state = []
        for action in possible_directions:
```

```
            x, y = state[0]
            dx, dy = Actions.directionToVector(action)
            nextx, nexty = int(x + dx), int(y + dy)
            if not self.walls[nextx][nexty]:
                valid_actions_from_state.append(action)
        return valid_actions_from_state

    def getActionCost(self, state, action, next_state):
        assert next_state == self.getNextState(state, action), (
            "Invalid next state passed to getActionCost().")
        return 1

    def getNextState(self, state, action):
        assert action in self.getActions(state), (
            "Invalid action passed to getActionCost().")
        x, y = state[0]
        dx, dy = Actions.directionToVector(action)
        nextx, nexty = int(x + dx), int(y + dy)
        nextFood = state[1].copy()
        nextFood[nextx][nexty] = False
        return ((nextx, nexty), nextFood)

    def getCostOfActionSequence(self, actions):
        """Returns the cost of a particular sequence of actions.  If those
actions
        include an illegal move, return 999999"""
        x,y= self.getStartState()[0]
        cost = 0
        for action in actions:
            # figure out the next state and see whether it's legal
            dx, dy = Actions.directionToVector(action)
            x, y = int(x + dx), int(y + dy)
            if self.walls[x][y]:
                return 999999
            cost += 1
        return cost

class AStarFoodSearchAgent(SearchAgent):
    "A SearchAgent for FoodSearchProblem using A* and your foodHeuristic"
    def __init__(self):
        self.searchFunction = lambda prob: search.aStarSearch(prob,
foodHeuristic)
        self.searchType = FoodSearchProblem

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 ***"
    foodGridList = foodGrid.asList()
    farthest_food = position
    farthest_distance = 0
    for food in foodGridList:
        if abs(position[0] - food[0]) + abs(position[1] - food[1]) >
farthest_distance:
            farthest_food = food
            farthest_distance = abs(position[0] - food[0]) + abs(position[1] -
food[1])
    diff = position[0] - farthest_food[0]
    count = 0
    for food in foodGridList:
        if diff > 0 and position[0] < food[0]:
            count += 1
        elif diff < 0 and position[0] > food[0]:
            count += 1
        elif diff == 0 and position[0] != food[0]:
            count += 1
    return farthest_distance + count

class ClosestDotSearchAgent(SearchAgent):
    "Search for all food using a sequence of searches"
    def registerInitialState(self, state):
        self.actions = []
        currentState = state
        while(currentState.getFood().count() > 0):
            nextPathSegment = self.findPathToClosestDot(currentState) # The
missing piece
            self.actions += nextPathSegment
            for action in nextPathSegment:
                legal = currentState.getLegalActions()
                if action not in legal:
                    t = (str(action), str(currentState))
                    raise Exception('findPathToClosestDot returned an illegal
move: %s!\n%s' % t)
                currentState = currentState.generateChild(0, action)
        self.actionIndex = 0
        print('Path found with cost %d.' % len(self.actions))

    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 ***"
        path = []
        foodGridList = food.asList()
        closest = foodGridList.pop()
        for f in foodGridList:
            if abs(f[0] - startPosition[0]) + abs(f[1] - startPosition[1]) <
abs(closest[0] - startPosition[0]) + abs(closest[1] - startPosition[1]):
                closest = f
        problem.goal = closest
        path = search.aStarSearch(problem, manhattanHeuristic)
        return path

class AnyFoodSearchProblem(PositionSearchProblem):
    """
    A search problem for finding a path to any food.

    This search problem is just like the PositionSearchProblem, but has a
    different goal test, which you need to fill in below.  The state space and
    child function do not need to be changed.

    The class definition above, AnyFoodSearchProblem(PositionSearchProblem),
    inherits the methods of the PositionSearchProblem.

    You can use this search problem to help you fill in the findPathToClosestDot
    method.
    """

    def __init__(self, gameState):
        "Stores information from the gameState.  You don't need to change this."
        # Store the food for later reference
        self.food = gameState.getFood()

        # Store info for the PositionSearchProblem (no need to change this)
        self.walls = gameState.getWalls()
        self.startState = gameState.getPacmanPosition()
        self.costFn = lambda x: 1
        self._visited, self._visitedlist, self._expanded = {}, [], 0 # DO NOT
CHANGE

    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

        "*** YOUR CODE HERE ***"

        return self.goal == state
```

```
def mazeDistance(point1, point2, gameState):
    """
    Returns the maze distance between any two points, using the search functions
    you have already built. The gameState can be any game state -- Pacman's
    position in that state is ignored.

    Example usage: mazeDistance( (2,4), (5,6), gameState)

    This might be a useful helper function for your ApproximateSearchAgent.
    """
    x1, y1 = point1
    x2, y2 = point2
    walls = gameState.getWalls()
    assert not walls[x1][y1], 'point1 is a wall: ' + str(point1)
    assert not walls[x2][y2], 'point2 is a wall: ' + str(point2)
    prob = PositionSearchProblem(gameState, start=point1, goal=point2,
warn=False, visualize=False)
    return len(search.bfs(prob))
```