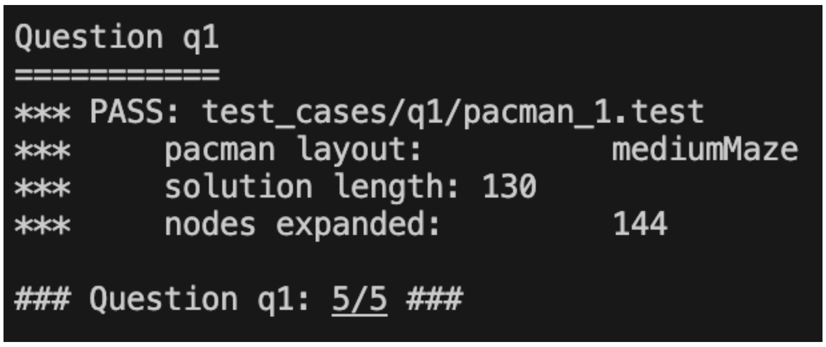
Show your autograder results and describe each algorithm:

Q1. Depth First Search (1%)



```
Question q1
===========
*** PASS: test_cases/q1/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 130
***     nodes expanded:         144

### Question q1: 5/5 ###
```

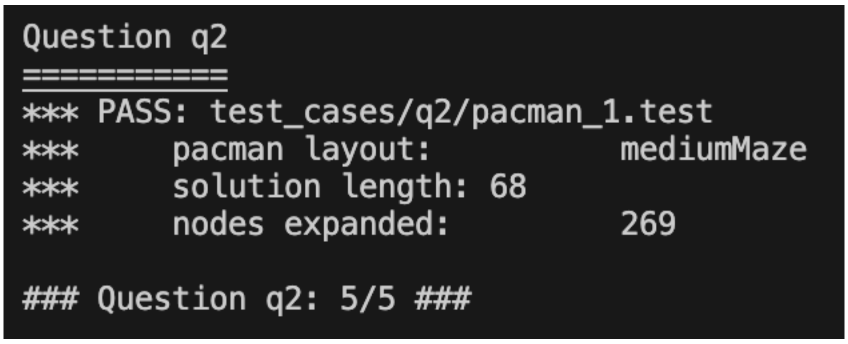
```
Let frontier be stack
frontier.push((start_state, []))
# Push the state and the actions taken to reach the state
mark state_state as explored
while frontier is not empty:
    state, actions = frontier.pop()
    if state == goal state then:
        return actions
    for all successors of the state:
        if successor is not explored then:
            frontier.push((successor, actions + action))
            #"action" is the action required to get to successor
            mark successor as explored
return
```

DFS 使用 stack 實現，按照它們添加的相反順序進行探索，因此探詢方向更多

是往深度探詢，透過 actions 我們去紀錄從初 state 到現在 state 所需要做的

action。搜尋直到 state == goal state.

Q2. Breadth First Search (1%)

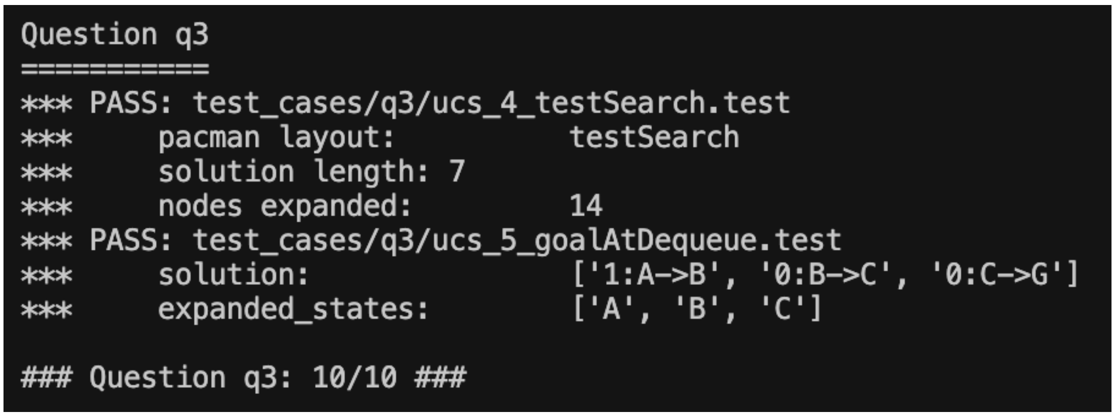


```
Question q2
===========
*** PASS: test_cases/q2/pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         269

### Question q2: 5/5 ###
```

```
Let frontier be queue
frontier.push((start_state, []))
# Push the state and the actions taken to reach the state
mark state_state as explored
while frontier is not empty:
    state, actions = frontier.pop()
    if state == goal state then:
        return actions
    for all successors of the state:
        if successor is not explored then:
            frontier.push((successor, actions + action))
            #"action" is the action required to get to successor
            mark successor as explored
return
```

BFS 使用 queue 實現，它從初始狀態開始，探索所有相鄰節點，然後移動到下一層節點（依照先進先出特性）。 透過 actions 我們去紀錄從初 state 到現在 state 所需要做的 action。搜尋直到 state == goal state.

Q3. Uniform Cost Search (1%)



```
Question q3
===========
*** PASS: test_cases/q3/ucs_4_testSearch.test
***     pacman layout:          testSearch
***     solution length: 7
***     nodes expanded:         14
*** PASS: test_cases/q3/ucs_5_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']

### Question q3: 10/10 ###
```

```
Let frontier be priorityqueue
#states with lower priority values have higher priority
frontier.push((start_state, [], 0), 0)
#Push ((state, a list of actions, cost),priority value) onto the priority queue
priorities[start_state] = 0 /
#record the priority values of the state
while frontier is not empty:
    state, actions, cost = frontier.pop()
    mark state as explored
    if state == goal state then:
        return actions
    for all successors of the state:
        new_cost = cost + step_cost
        #Calculate the new cost to reach the successor,step_cost is the incremental cost of expanding to that successor.
        if successor not explored then :
            if successor not in priorities then:
            #the successor If it hasn't been encountered before
                frontier.push((successor, actions + [action], new_cost), new_cost)
            elif new_cost is lower priorities[successor] then:
            #the successor has been encountered before but should update the priority value
                frontier.update((successor, actions + [action], new_cost), new_cost)
            priorities[successor] = new_cost
    return
```

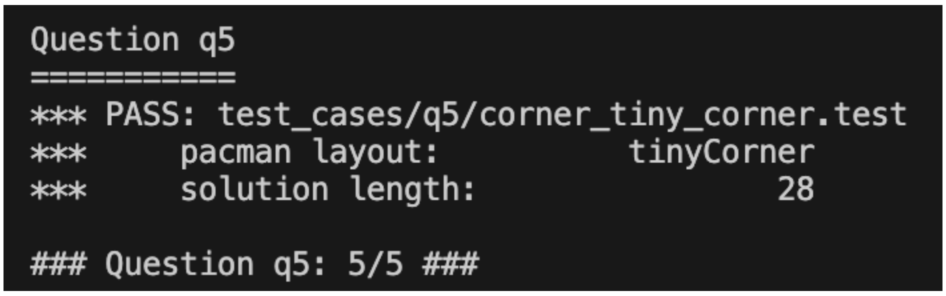
UCS，是一種基於成本的搜尋算法。它使用 priorityque，並按照成本順序搜索（成本越低越優先），並隨時更新已探索狀態的最低成本（存於 frontier 的 state），搜尋直到 state == goal state.。

Q4. A* Search (null Heuristic) (1%)

```
Let frontier be a priority queue
#where states with lower priority values have higher priority
frontier.push((start_state, [], 0), 0 + heuristic(start_state, problem))
#Push ((state, a list of actions, cost), priority value) onto the priority queue. Since it is null heuristic, can simply remove the utilization of heuristic
priorities[start_state] = 0 + heuristic(start_state, problem)
while frontier is not empty:
    state, actions, cost = frontier.pop()
    mark state as explored
    if state == goal state:
        return actions
    for all successors of the state:
        total_cost = cost + step_cost
        # Calculate the new cost to reach the successor, where step_cost is the incremental cost of expanding to that successor.
        priority_cost = total_cost + heuristic(successor, problem)
        if successor not explored then:
            if successor not in priorities:
            # If the successor hasn't been encountered before
                frontier.push((successor, actions + [action], total_cost), priority_cost)
            elif new_cost is lower than priorities[successor]:
            #If the successor has been encountered before but should update the priority value
                frontier.update ((successor, actions + [action], total_cost), priority_cost)
            priorities[successor] = total_cost
return
```

A* Search 和 USC 類似，同樣使用 priorityque。然而 A* Search 使用了 priority value 為實際代價和當前狀態到達目標狀態的代價（然而在這一個函數並沒有差別，因為是 null heuristic），這使得 A 搜索在每一步擴展時都考慮了到目標狀態的預期代價。而 UCS 僅僅使用實際代價來決定優先順序，並沒有考慮目標狀態的估計代價。同樣搜尋直到 state == goal state.。

Q5. Breadth First Search (Finding all the Corners) (1%)



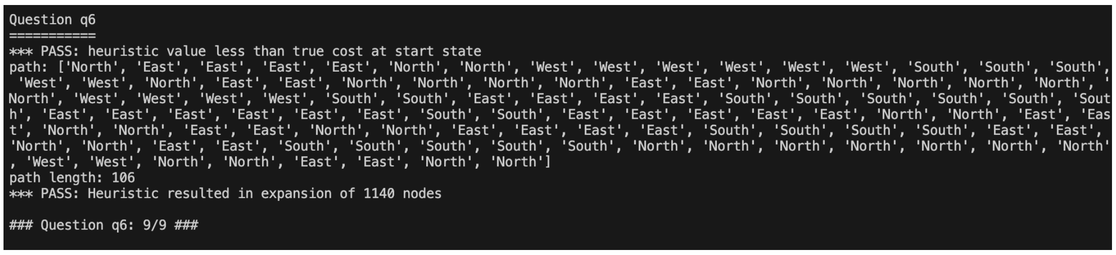
```
Question q5
===========
*** PASS: test_cases/q5/corner_tiny_corner.test
***     pacman layout:          tinyCorner
***     solution length:                28

### Question q5: 5/5 ###
```

```
getStartState(self):
    return (self.startingPosition , self.corners)
isGoalState(self, state: Any):
    return len(list(state[1])) == 0 all corners were visited
getSuccessors(self, state: Any):
    successors = []
    for all action in (north, south, east, west):
        nextState = position+ action
        #current position (x, y) from the state + the action's vector
        if nextState is not wall:
            unvisited_corners = list(state.unvisited_corners)
            if nextState is in unvisited_corners then:
                unvisited_corners.remove(nextState)
            successors.append(((nextState, tuple(unvisited_corners)), action, 1))
    self._expanded += 1
    return successors
```

每一個 state 包含位置以及未訪問的角落列表，對於 sucessor 的存取，搜索 state 的四個方位為 nextState，假設 nextState 不是牆壁，我們計算行經 nextState 後所剩餘的未訪問的角落，並將其 append 到 successors 中。透過這一樣的算法，我們可以發現當 nextState 等於 corner 時，隨後 state = nextState 時，其四周包含其 parents(行經到 nextState 的所有 state)的未訪問的角落列表都會少一個 corner。

當未訪問的角落列表為空，代表四個角落都已經行經。

Q6. A* Search (Corners Problem: Heuristic) (1%)



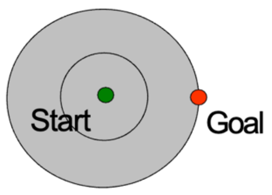
```
Question q6
===========
*** PASS: heuristic value less than true cost at start state
path: ['North', 'East', 'East', 'East', 'East', 'North', 'North', 'West', 'West', 'West', 'West', 'West', 'West', 'South', 'South', 'South',
 'West', 'West', 'North', 'East', 'East', 'North', 'North', 'North', 'North', 'East', 'East', 'North', 'North', 'North', 'North', 'North', '
North', 'West', 'West', 'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'South', 'Sout
h', 'East', 'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North', 'East', 'Eas
t', 'North', 'North', 'East', 'East', 'North', 'North', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'East', 'East',
'North', '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 1140 nodes

### Question q6: 9/9 ###
```

```
if all corners are visited then :
return 0
finding the maximum Manhattan distance ( max_distance) between the current
position and any unvisited corners
return max_distance
```

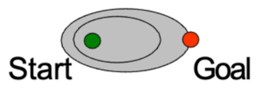
我試過三個做法來實行，其中一個是計算探詢所有未被訪問角落的最小曼哈頓距離。另一個是計算現在位置到任一未被訪問角落的最小曼哈頓距離。但兩這效益都沒有計算現在位置到任一未被訪問角落的最大曼哈頓距離好。

Describe the difference between Uniform Cost Search and A* Contours (2%)

- Uniform-cost expands equally in all "directions"



- A* expands mainly toward the goal, but does hedge its bets to ensure optimality



UCS 可以被視為 null heuristics 的，或者所有 state 的 heuristic value 都相同的 A*。UCS 的 search priority 僅取決於從起始節點到當前節點的實際路徑成本，它將以一圈一圈的方式探索搜索空間及擴展節點，直到找到目標。因此 UCS 的 expanded contour 更接近圓形。

而 A * 來說，靠近 goal 的地區其 heuristic value 通常更小，這演算法更傾向搜索接近目標的 state。這使得 A 的擴展輪廓更接近橢圓形，因為在搜索過程中，它會更快地向目標方向前進，而不是像 UCS 一樣以均勻的方式探索搜索空間。

Describe the idea of Admissibility Heuristic (2%)

A heuristic h is admissible (optimistic) if:
$0 <= h(n) <= h^*(n)$
where is $h^*(n)$ the true cost to a nearest goal

Admissibility Heuristic value 都必須小於或等於從該狀態到達目標的實際成本。