人工智能 Project1

Q1：Finding a Fixed Food Dot using Depth First Search

1、 问题描述

编写 search.py 中 depthFirstSearch 函数，通过深搜找到 goal，并返回到达 goal 的 action 数组

首先阅读源代码，与本题有关的重点内容如下：

有关数据结构使用：深搜采用 stack 实现，相关数据结构在 util.py 可查阅

```
class Stack:
    "A container with a last-in-first-out (LIFO) queuing policy.
    def __init__(self):
        self.list = []

    def push(self,item):
        "Push 'item' onto the stack"
        self.list.append(item)

    def pop(self):
        "Pop the most recently pushed item from the stack"
        return self.list.pop()

    def isEmpty(self):
        "Returns true if the stack is empty"
        return len(self.list) == 0
```

函数输入参数类分析：SearchProblem

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

获取起始状态，作为深搜第一个入栈元素

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

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

判断是否到达目标状态，深搜结束标志

```
def getSuccessors(self, state):
    """
      state: Search state

    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.
    """
    util.raiseNotDefined()
```

相关状态搜索函数，返回一个元组，内含三个参数，分别为下一个相关状态、到达该状态需要的 action、步长（cost）

利用上面三个函数可以实现深搜，思路为：将初始状态入栈，每次遍历 pop 栈顶状态，判断是否到达 goal，到达返回 action 路径，否则标记已访问并 push 所有相关状态，搜索路径+=前往新入栈状态的 action

2、 算法设计及实现

```
depthPath, isVisited=[], []
stack = util.Stack()
startState = problem.getStartState()
startPath = [] #初始状态和初始路径
stack.push((startState, startPath))
```

创建深搜路径和访问数组，前者用来存储每次前往目标状态需要的 action，后者用来标记已访问的结点（状态）。

采用数据结构 stack，通过 getStartState 函数获取起始状态，放入栈顶

```
while not stack.isEmpty():
    (State, Path) = stack.pop()
    if problem.isGoalState(State): #已经到达目的地
        depthPath = Path
        break
```

在栈非空情况下每次遍历获取栈顶元素，分别存到状态和路径中。利用 isGoalState 判断是否到达目的地。

```
if State not in isVisited:
    # 打已访问标记
    isVisited.append(State)
    for nextState, nextPath, _ in problem.getSuccessors(State):
        newPath = Path + [nextPath]
        # 拓展节点入栈
        stack.push((nextState, newPath))
```

如果没到达，且该节点没被访问过，则加入访问数组标记，并对该状态调用 getSuccessors 返回下一步，到达下一步的 action 和 cost（此问没用）。将下一步 action 加入到达栈顶路径数组 Path 中作为到达下一状态的路径数组，与下一状态打包入栈重新循环。最后退出循环返回 depthPath 数组

3、 实验结果

```
(search) E:\人工智能\search\search>python pacman.py -l tinyMaze -p SearchAgent
[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
Ending graphics raised an exception: 0
Average Score: 500.0
Scores:        500.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
(search) E:\人工智能\search\search>python pacman.py -l mediumMaze -p SearchAgent
[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
```

```
(search) E:\人工智能\search\search>python pacman.py -l bigMaze -z .5 -p SearchAgent
[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
```

```
(search) E:\人工智能\search\search>python autograder.py -q q1
Starting on 4-1 at 15:07:08

Question q1
===========
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'D', 'C']
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:               ['2:A->D', '0:D->G']
***     expanded_states:        ['A', '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:               ['2:A->B2', '0:B2->C', '0:C->D', '2:D->E2', '0:E2->F', '0:F->G']
***     expanded_states:        ['A', 'B2', 'C', 'D', 'E2', 'F']
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 130
***     nodes expanded:         146

### Question q1: 3/3 ###
```

Q2：Breadth First Search

1、 问题描述

编写 search.py 中 breadthFirstSearch 函数，通过广搜找到 goal，并返回到达 goal 的 action 数组

从数据结构角度思考，广搜是一层一层往下搜索，所以选择 queue（FIFO），在深搜的基础上改动，每次出队元素判断 goal 后，入队所有相连元素，从而保证一层搜完再搜下一层

查看 util.py 中 queue 定义

```
class Queue:
    "A container with a first-in-first-out (FIFO) queuing policy."
    def __init__(self):
        self.list = []

    def push(self,item):
        "Enqueue the 'item' into the queue"
        self.list.insert(0,item)

    def pop(self):
        """
          Dequeue the earliest enqueued item still in the queue. This
          operation removes the item from the queue.
        """
        return self.list.pop()

    def isEmpty(self):
        "Returns true if the queue is empty"
        return len(self.list) == 0
```

算法整体框架与深搜差别不大，所用到的函数同 Q1

2、 算法设计与实现

```
breadthPath, isVisited = [], []
queue = util.Queue()
startState = problem.getStartState()
startPath = []
queue.push((startState, startPath))
```

创建广搜路径（最后返回的）和访问数组，创建队列，入队初始状态及初始路径（到达初始状态的 action 数组）

```
while not queue.isEmpty():
    (State, Path) = queue.pop()
    if problem.isGoalState(State):
        breadthPath = Path
        break
```

队列非空循环取队首元素，判断队首元素状态是否为 goal

```
if State not in isVisited:
    isVisited.append(State)
    for nextState, nextPath, _ in problem.getSuccessors(State):
        newPath = Path + [nextPath]
        queue.push((nextState, newPath))
```

如果未访问且非 goal，则标记状态，查找相关可访问状态，返回下一个可转移状态和到该状态的 action，将 action 加到路径数组，和下一个状态打包入队。

3、 实验结果

```
(search) E:\人工智能\search\search>python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs
[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
```

```
(search) E:\人工智能\search\search>python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5
[SearchAgent] using function bfs
[SearchAgent] using problem type PositionSearchProblem
Path found with total cost of 210 in 0.1 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
```

```
After 5 moves: left
-------------
|   | 1 | 2 |
-------------
| 3 | 4 | 5 |
-------------
| 6 | 7 | 8 |
-------------
Press return for the next state...
```

```
(search) E:\人工智能\search\search>python autograder.py -q q2
Starting on 4-1 at 15:20:26

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: 3/3 ###


Finished at 15:20:26

Provisional grades
==================
Question q2: 3/3
------------------
Total: 3/3
```

Q3：Varying the Cost Function

1、 问题描述

本题要求在 BFS 的基础上，考虑成本最低的路径，找到 goal 后并返回路径数组

考虑到路径成本最低，需要计算每走一步带来的 cost，如果每走一步都能选择接近 goal 的 cost 最小的路径，那么最终到达 goal 的路径总 cost 理论上最小。

综上所述，在广搜的基础上考虑 cost 最小，可以选择优先队列，通过维护一个队首元素 cost 最小的队列来实现。

首先观察 util.py 中对优先队列的类定义

```
class PriorityQueue:

    def __init__(self):
        self.heap = []
        self.count = 0

    def push(self, item, priority):
        entry = (priority, self.count, item)
        heapq.heappush(self.heap, entry)
        self.count += 1

    def pop(self):
        (_, _, item) = heapq.heappop(self.heap)
        return item

    def isEmpty(self):
        return len(self.heap) == 0
```

这里 push 函数除了 self 和 item 外多了一个 priority 参数，即根据传入的 priority 来决定堆顶元素，维护是通过 heappush 实现，定义具体如下：

```
def heappush(heap, item):
    """Push item onto heap, maintaining the heap invariant."""
    heap.append(item)
    _siftdown(heap, 0, len(heap)-1)
```

2、 算法设计及实现

思路：参考 BFS 实现原理，在入队出队中加入 cost 权值，采用优先队列实现

代码大体上同 Q2，其中做出以下改变

```
startCost = 0
heap.push((startState, startPath, startCost), startCost)
```

记录初始状态的 cost（设为 0），入队元素除了状态，路径，成本（打包为一个 item）外还有一个 cost（priority）

```
for nextState, nextPath, nextCost in problem.getSuccessors(State):
    newPath = Path + [nextPath]
    newCost = Cost + nextCost
    heap.push((nextState, newPath, newCost), newCost)
```

从 getSuccessors 中接受第三个参数(前往下一个状态所需 cost)并加和到总 Cost 中，作为下一个入队元素的 priority

3、 实验结果

```
(search) E:\人工智能\search\search>python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs
[SearchAgent] using function ucs
[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
```

```
(search) E:\人工智能\search\search>python pacman.py -l mediumDottedMaze -p StayEastSearchAgent
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
```

```
(search) E:\人工智能\search\search>python pacman.py -l mediumScaryMaze -p StayWestSearchAgent
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
```

```
Provisional grades
==================
Question q3: 3/3
------------------
Total: 3/3
```

Q4：A* search

1、问题描述

采用 A*搜索算法找到 goal，并返回路径数组

A*搜索本质上跟 usc 差不多，区别在于 A*多一个启发式函数，用来估计当前状态到下一个状态的 cost，在上一问中实现的 usc 搜索，维护优先队列的参数是真实值 cost，在 A*搜索中除了真实值 cost 还需要参考启发式函数返回值 cost。即两者之和越小，越应该在队前方。

这一文中所用的启发式函数题提供了 manhattanHeuristic 距离启发函数，因此不需要我们考虑，只需要同上一问维护一个优先队列即可。

2、算法设计及实现

整体思路同 Q3，需要做出改变的地方如下：

```
heap.push((nextState, newPath, newCost), newCost + heuristic(nextState, problem))
```

在每次入队新状态时，item 内容不变，但 priority 除了 getSuccessors 返回的真实值 cost 外还需要加上启发式函数返回值 cost

这里 heuristic 选择了 manhattanHeuristic 距离启发函数，具体参数如下

```
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
```

第一个参数为下一步的状态，第二个参数为问题本身。

曼哈顿距离启发函数由 python 代码传递

```
python pacman.py -l bigMaze -z .5 -p SearchAgent -a
fn=astar,heuristic=manhattanHeuristic
```

3、实验结果

```
(search) E:\人工智能\search\search>python pacman.py -l bigMaze -z .5 -p
[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
```

```
Provisional grades
==================
Question q4: 3/3
------------------
Total: 3/3
```

Q5：Finding All the Corners

1、问题描述

采用已完成的 bfs 搜索算法，对迷宫内四个角进行搜索，即返回一条路径，经过四个角落

这一题用到之前的 bfs 搜索算法，在不改变搜索算法本身的情况下，我们需要做出改变的函数可能有：getStartState，isGoalState，getSuccessors

```
def breadthFirstSearch(problem: SearchProblem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    breadthPath, isVisited = [], []
    queue = util.Queue()
    startState = problem.getStartState()
    startPath = []
    queue.push((startState, startPath))
    while not queue.isEmpty():
        (State, Path) = queue.pop()
        if problem.isGoalState(State):
            breadthPath = Path
            break
        if State not in isVisited:
            isVisited.append(State)
            for nextState, nextPath, _ in problem.getSuccessors(State):
                newPath = Path + [nextPath]
                queue.push((nextState, newPath))
    return breadthPath
```

初步分析，我们需要确定任务完成的定义，即 goal 的判定。题目需要访问所有四个角，所以 isGoalState 判断条件应该为四个角都被访问过，否则继续循环

其次考虑已经被访问过的角不再去靠拢，即 getSuccessors 中返回下一步状态不应该向已经访问过的角落去靠近，所以该函数需要做改动

最后由于两个函数输入都是 state，因此 getStartState 返回值也需要做改动，需要实时知道 pacman 的当前坐标和地图中剩余未被访问过的角落或者已经访问过的角落，避免死循环

2、算法设计及实现

```
def getStartState(self):
    """
    Returns the start state (in your state space, not the full Pacman state
    space)
    """
    "*** YOUR CODE HERE ***"
    # 初始状态
    startState = self.startingPosition
    # 已访问的角落
    visitedCorners = []
    return startState, visitedCorners
    util.raiseNotDefined()
```

获取初始状态，创建已访问角落数组，并一起返回

```
def isGoalState(self, state: Any):
    """
    Returns whether this search state is a goal state of the problem.
    """
    "*** YOUR CODE HERE ***"
    # 每个角落都访问过了
    return len(state[1]) == 4
    util.raiseNotDefined()
```

判断 goal 条件：四个角落都被访问过。

这里由于 state 实际上含两个元素，state[0]为坐标，state[1]为 visitedCorners 数组，对其求 len，等于四说明四个均已访问

```
x, y = state[0]
visitedCorners = state[1]
dx, dy = Actions.directionToVector(action)
nextx, nexty = int(x + dx), int(y + dy)
hitswall = self.walls[nextx][nexty]
nextNode = (nextx, nexty)
```

getSuccessors 中首先获取当前状态坐标和已访问的角落，然后根据注释内容，调用函数找到下一个坐标点。

```
if not hitswall: # 加入不撞墙的状态
    visitedCornersList = list(visitedCorners)
    if nextNode in self.corners:
        if nextNode not in visitedCornersList:
            visitedCornersList.append(nextNode)
```

对下一个坐标点判断是否为墙，不是则创建已访问角落列表，对是地图角落且不在已访问列表中的角落入队

```
newState = (nextNode, visitedCornersList)
successors.append((newState, action, 1))
```

将新坐标和新的已访问数组打包作为新状态，和走到这个状态的 action 即路径和 cost 打包传回。

3、实验结果

```
(search) E:\人工智能\search\search>python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[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
```

```
(search) E:\人工智能\search\search>python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem
[SearchAgent] using function bfs
[SearchAgent] using problem type CornersProblem
Path found with total cost of 106 in 0.2 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
```

```
Provisional grades
==================
Question q2: 3/3
Question q5: 3/3
------------------
Total: 6/6
```

Q6：Corners Problem: Heuristic

1、问题描述

在 Q5 的基础上，要求用启发式函数降低 cost。

根据提供运行测试代码发现，测试用例为 AStarCornersAgent，启发式函数为哈曼顿，所以本题需要确定一个启发的方向。

2、算法设计及实现

我的想法是再剩余所有角落中尽可能往远的角落走，因为既可以避免去访问很近的而拉远了其余的角落也说不定顺路就遇到未访问的角落。至于怎么往远走，我通过对剩余未访问过的角落进行平均距离计算，选择平均距离最远的角落

```
visited_corners = state[1]
remaining_corners = []
for corner in corners:
    if corner not in visited_corners:
        remaining_corners.append(corner)
```

创建剩余角落数组，如果没有则置为 0，如果还有对每个角落求哈曼顿距离，最后平均距离

```
if not remaining_corners:
    return 0
distances = [util.manhattanDistance(state[0], corner) for corner in remaining_corners]
avg_distance = sum(distances) / len(distances)
return avg_distance
```

3、实验结果

```
(search) E:\人工智能\search\search>python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5
Path found with total cost of 106 in 0.1 seconds
Search nodes expanded: 1540
Pacman emerges victorious! Score: 434
Average Score: 434.0
Scores:        434.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
Provisional grades
==================
Question q4: 3/3
Question q6: 2/3
------------------
Total: 5/6
```

Q7：Eating All The Dots

1、问题描述

类似找角落问题，只不过把四个角落扩大到未知个食物，注意到这里 state 为两部分，一是 pacman 的坐标，二是所处食物地图。

2、算法设计及实现

类似找角落，同样选取剩余所有食物距离的平均值作为启发

```
position, foodGrid = state
"*** YOUR CODE HERE ***"
foodList = foodGrid.asList()
if not foodList:
    return 0
distances = [mazeDistance(position, foodPosition, problem.startingGameState) for foodPosition in foodList]
avg_distance = sum(distances) / len(distances)
return avg_distance
```

3、实验结果

```
(search) E:\人工智能\search\search>python pacman.py -l testSearch -p AStarFoodSearchAgent
Path found with total cost of 7 in 0.0 seconds
Search nodes expanded: 11
Pacman emerges victorious! Score: 513
Ending graphics raised an exception: 0
Average Score: 513.0
Scores:        513.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
Provisional grades
==================
Question q4: 3/3
Question q7: 4/4
------------------
Total: 7/7
```

Q8：Suboptimal Search

1、 问题描述

找次优解，每次吃完最近的食物再去找下一个最近的食物。

这里直接沿用前面写的 ucs 寻路，因为本质是广搜，一层一层相当于每次先考虑周围一圈，这样能发现最近的食物

2、算法设计及实现

首先修改 goal，只要当前位置是食物列表中的，就为 true

```
"*** YOUR CODE HERE ***"
isGoal = False
if (x,y) in self.food.asList():
    isGoal = True
return isGoal
util.raiseNotDefined()
```

寻路算法照搬

3、实验结果

```
(search) E:\人工智能\search\search>python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
Path found with cost 350.
Pacman emerges victorious! Score: 2360
Average Score: 2360.0
Scores:        2360.0
Win Rate:      1/1 (1.00)
Record:        Win
```

```
Provisional grades
==================
Question q8: 3/3
------------------
Total: 3/3
```

总结

本次实验虽然从结果上看绝大多数都过了测试，但是从游戏 UI 界面发现，有时候 pacman 会忽略眼前仅剩一两个的点去绕远的吃，等等很多类似的问题会出现，例如 dfs 不能找到最短路径，bfs 可以找到最短但是内存占用大会卡，usc 在 cost 大量为 1 或相同情况下也无能为力，astar 算法看上去最合理，但是很依赖所选的启发式函数，无论是优先考虑较远的角落还是平均距离，不同场景下不同启发效果也不一样，所以通过这些算法实例，应用场景的关系很大。