一．简介/问题描述

1.1 待解决问题的解释

本实验用一个叫 pacman 的游戏让学生编写一些搜索策略控制吃豆人去吃豆子。吃豆人在行走的过程中需要遵守游戏规则，如不能翻墙、不能碰到幽灵等。该游戏以找到解决方案需要的时间、拓展的节点数等为评分指标，对于吃豆人的搜索路径给出评价。

本实验共有 8 个问题，其中我写的是问题三的代价一致算法和问题八的次最优搜索。由于问题八需要用到其他的一些搜索算法，所以对于其他的问题我也有所描述。

1.2 问题的形式化描述

状态定义：

Pos(x,y)：pacman 位于点(x,y)上

destination(x,y): 点(x,y)是目标的点]

wall(x,y): 点(x,y)是墙，无法通过

动作定义:

gWest: 向西走一步

条件: !wall(x-1,y) && pos(x,y)

结果: 移除 pos(x,y)，添加 pos(x-1,y)

gEast: 向东走一步

条件: !wall(x+11,y) && pos(x,y)

结果: 移除 pos(x,y), 添加 pos(x+1,y)

gSouth: 向南走一步

条件: !wall(x,y+1) && pos(x,y)

结果: 移除 pos(x,y), 添加 pos(x,y+1)

gNorth: 向北走一步

条件: !wall(x,y-1) && pos(x,y)

结果: 移除 pos(x,y), 添加 pos(x,y-1)

起始状态:

! Pos(x, y) && destination(x, y)

终止状态:

Pos(x, y) && destination(x, y)

1.3 解决方案介绍（原理）

问题一:

使用深度优先算法拓展结点，使用栈作为 Open 表的数据结构。深度优先算法以搜索树为原型，每一次选择深度最大的节点进行拓展，是后生成的节点先拓展的策略。一般不能保证找到最优解。

问题二：

使用广度优先算法拓展结点，广度优先算法每一次将同样深度的节点按一定顺序逐个进行拓展，是一种先生成的节点先拓展的策略。因此使用队列作为 open 表的数据结构。一定能够找到最优解，但是往往搜索效率较低。

问题三：

使用代价一致算法算法进行扩展。代价一致算法是广度优先算法的推广，使用优先队列存储拓展的节点，每一次选择队列中代价最小的节点进行拓展。同样也一定能找到最优路径，并且比广度优先算法更快地找到最优路径。

问题四：

使用 A*算法进行扩展。A*算法是对 A 算法的估价函数 f (n)=g(n)+h(n)加上某些限制后得到的一种启发式搜索算法，经过限制后，新的估价函数为 f *(n)=g *(n)+h*(n)。其中，g *(n)表示从开始节点到节点 n 的最小代价，h*(n)表示从节点 n 到目标节点的最小代价，通过对二者进行估计，进而 得出最优路径实际代价 f *(n)的估计值 f (n) ，结合贪心 算法选择具有全局最小 f (n)的节点进行拓展，直到找到目标节点。

问题五~问题八：

基于以上的四种搜索算法稍作修改后进行应用。

二．算法介绍

2.1 所用方法的一般介绍及伪代码

问题 1：应用深度优先算法找到一个特定的位置的豆

用栈作为 open 表存储拓展到的结点，用 closed 表来对已经遍历过的位置进行标记。如果栈不为空，则取出栈顶未被遍历过的节点，将其未被遍历过的子节点(含位置、方向内容）进行压栈，一直到找到目标豆子的位置。

伪代码：

（1）把初始节点 S0 放入 Open 表，建立一个 CLOSED 表，置为空；

（2）检查 Open 表是否为空表，若为空，则问题无解，失败退出；

（3）把 Open 表的第一个节点取出放入 Closed 表，并记该节点为 n；

（4）考察节点 n 是否为目标节点，若是则得到问题的解成功退出；

（5）若节点 n 不可扩展，则转第（2）步；

（6）扩展节点 n， 将其子节点放入 Open 表的首部，并为每个子节点设置指向父节点的指针，转向第（2）步。

问题 2：广度优先算法

广度优先算法与深度优先算法唯一的不同是 Open 表的数据结构为队列。由于队列具有先进先出的特性 ,因此每一次都会逐层拓展节点 ,达到广度优先的目的。因此伪代码不再重复放入。

问题 3：代价一致算法

与广度优先算法类似，只不过 Open 表的数据结构由队列变为了优先队列，并且是按代价为键值进行排序的。因此每次先出队列的一定是代价最小的结点，从而达到代价一致算法的目的。因此伪代码也不再重复放入。

问题 4：A* 算法

A*算法如果使用的启发式函数 h*（n）为 0，就退化成了代价一致算法。这里取当前点距目标点的曼哈顿距离作为启发式函数。满足 A*算法的最优性和单调性。令 g*（n）为起点到该点的距离，则 f*（n）=g*（n）+h*（n）作为代价,与代价一

致算法类似地将节点数据放入以优先队列作为数据结构的 Open 表中,其余操作均与代价一致算法一样。因此伪代码也不再重复放入。

问题 7：吃掉所有的豆子

通过 A*算法寻找路径。

问题 8：次最优搜索

这题是需要优先吃最近的豆子，因此 A*、bfs、ucs 都可以解决“寻找最近的豆子”的问题，这里我选择使用 A*，可以使搜索树更小。

（1）准备工作：得到起始位置和食物、墙的位置

（2）定义问题 AnyFoodSearchProblem(gameState)

（3）返回用 A*算法找到的上述问题的最短路径

isGoalState(self, state):

（1） 取出坐标值(x, y)

（2） 得到食物位置 foodGrid

（3） if 该坐标处有食物或者当前地图中已经没有食物：

（4） Return True

三．算法实现

3.1 实验环境与问题规模

实验环境： ubuntu20.04

pycharm professional edition 2020.2

Python 3.8

问题规模： 本实验为基于搜索树的最优路径搜索问题。由于地图的限制，搜索树的每个节点最多只有 3 个子节点，代表能够行走的 3 个方向（已经访问过的节点不考虑）。设地图规模为 m*n，对吃豆人和 a 颗豆子， 共(m*n)^(1+a)种状态。

3.2 数据结构

深度优先： 栈

广度优先： 队列

代价一致算法和 A*： 优先队列

3.3 实验结果

代价一致算法

```
C:\Users\fay\Documents\Github\AI_lesson_labs\search-python3>python3 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
```

```
C:\Users\fay\Documents\Github\AI_lesson_labs\search-python3>python3 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
```

```
C:\Users\fay\Documents\Github\AI_lesson_labs\search-python3>python3 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
```

次最优搜索

```
C:\Users\fay\Documents\Github\AI_lesson_labs\search-python3>python3 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
```

四．总结及讨论（对该实验的总结以及任何该实验的启发）

该实验主要考察了学生对于搜索策略的了解程度，并且要求学生在一个实际的问题中实现他们。对于我来说，通过这次实验我对于 A*、BestFirst、ucs 等搜索算法有了更加深刻的了解。这些算法的相似度较高，总体的来说，A*是最好的，考虑到了已有的 cost 和对未来 cost 的估计，而 bestfirst 只考虑了对未来 cost 的估计，ucs 只考虑了已有的 cost。这也是算法不断改进的过程。

附录—源代码及其注释（纸质版不需要打印）

```
def breadthFirstSearch(problem):
    """Search the shallowest nodes in the search tree first."""
    "*** YOUR CODE HERE ***"
    from game import Directions
    STATE = 0
    ACTION = 1
    PRE_STATE = 2

    initState = problem.getStartState()

    queue = util.Queue()
    visited = util.Counter()

    # Every node is a triple, (nodeState, action, preNode) where
    # 'nodeState' is the state for current node,
    # 'action is how previous node goes to current node,
    # 'preNode' is the previous node
    queue.push((initState, "None", None))
    visited[initState] = 1

    lastNode = None
    while not queue.isEmpty():
        curr = queue.pop()

        if problem.isGoalState(curr[STATE]):
            lastNode = curr
            break
        else:
            nexts = problem.getSuccessors(curr[STATE])
            for node in nexts:
                if visited[node[0]] == 0:

                    #  未访问过

                    visited[node[0]] = 1
                    queue.push((node[0], node[1], curr))

    result = []
    if lastNode is not None:
        while lastNode[PRE_STATE] is not None:
            result.append(lastNode[ACTION])
            lastNode = lastNode[PRE_STATE]

    result.reverse()
    return result
```

```
def uniformCostSearch(problem):
    """Search the node of least total cost first."""
    "*** YOUR CODE HERE ***"
    from game import Directions
    STATE = 0
    ACTION = 1
    COST = 2
    PRE_STATE = 3

    initState = problem.getStartState()

    queue = util.PriorityQueueWithFunction(lambda e: e[COST])
    visited = util.Counter()

    # Every node is a triple, (nodeState, action, cost, preNode) where
    # 'nodeState' is the state for current node,
    # 'action is how previous node goes to current node,
    # 'preNode' is the previous node
    queue.push((initState, "None", 0, None))
    visited[initState] = 1

    lastNode = None
    while not queue.isEmpty():
        curr = queue.pop()

        if problem.isGoalState(curr[STATE]):
            lastNode = curr
            break
        else:
            nexts = problem.getSuccessors(curr[STATE])
            for node in nexts:
                if visited[node[0]] == 0:

                    #  未访问过

                    visited[node[0]] = 1
                    queue.push((node[0], node[1], node[2], curr))

    result = []
    if lastNode is not None:
        while lastNode[PRE_STATE] is not None:
            result.append(lastNode[ACTION])
            lastNode = lastNode[PRE_STATE]

    result.reverse()
```

```
    return result



    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.generateSuccessor(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)

            return search.aStarSearch(problem)          #调用 A*算法，寻找最短
路径
```