**同济大学计算机系**

**人工智能课程设计综合实验报告1**

****

**实验名称**  **Search**

**学 号**

**姓 名**

**专 业 计算机科学与技术**

**授课老师 武妍老师**

**完成日期 2024年4月7日**

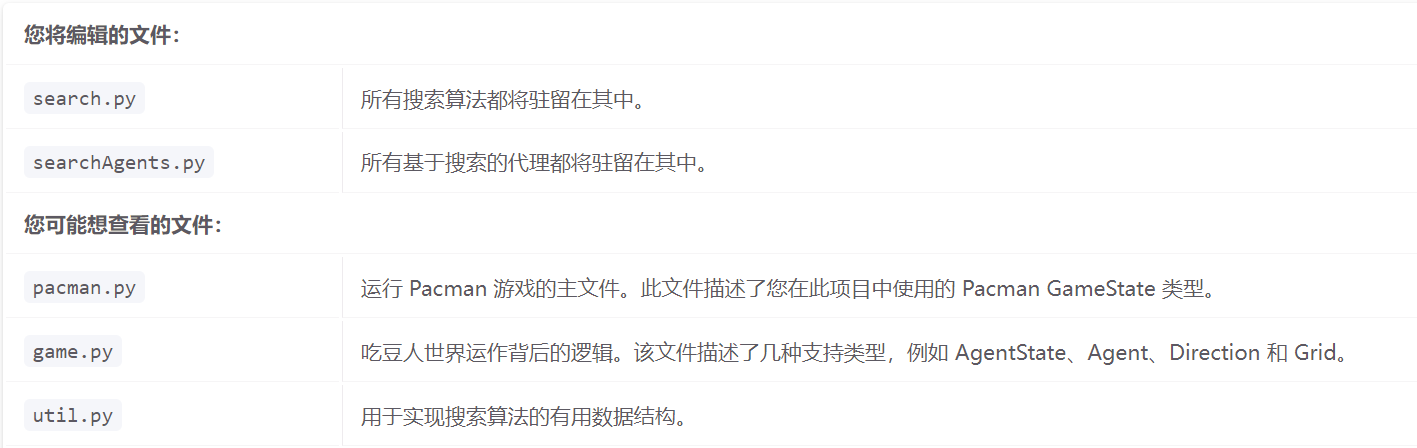
1. 项目概述与分析

**1、项目的总体概述**

吃豆人生活在一个闪亮的蓝色世界里，那里有蜿蜒的走廊和美味的圆形食物，我们需要帮助吃豆人有效率地到达某个地方或者吃掉食物并且避开危险。

本项目主要是设计程序来控制吃豆人搜索出到达特定目标的路径，途中收集食物并且避开怪物，以期获得更高的分数。本项目中有不同的地图资源，同时也有不同的搜索算法要求，我们需要设计不同搜索算法来解决这些问题。

**2、项目已有代码分析**

****

**searchAgent.py:**这个文件里包含了游戏各种模式对应的类和类的实现，以辅助实现这些类的一些函数和类；

class SearchAgent(Agent):使用指定的搜索算法解决一个指定的搜索问题

class PositionSearchProblem(search.SearchProblem):包含从起始点到终点的游戏模式所需的各种属性和函数；

manhattanHeuristic(position, problem, info={}):实现游戏模式1的一个启发式函数；

euclideanHeuristic(position, problem, info={}):实现游戏模式1的另一个启发式函数

class CornersProblem(search.SearchProblem): 包含遍历迷宫四个顶点的游戏模式所需的各种属性和函数

cornersHeuristic(state: Any, problem: CornersProblem): 实现游戏模式2的一个启发式函数

class FoodSearchProblem: 包含吃掉迷宫里所有食物的游戏模式所需的各种属性和函数

foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem): 实现游戏模式2的一个启发式函数

class ClosestDotSearchAgent(SearchAgent):游戏模式3的贪婪搜索算法

class AnyFoodSearchProblem(PositionSearchProblem): 包含吃掉迷宫里任何指定食物的游戏模式所需的各种属性和函数

mazeDistance(point1: Tuple[int, int], point2: Tuple[int, int], gameState: pacman.GameState) ->int:计算任意两个点之间的距离

**search.py:**包含完成各种游戏模式的搜索算法

depthFirstSearch(problem: SearchProblem):深度优先搜索算法

breadthFirstSearch(problem: SearchProblem):宽度优先搜索算法

uniformCostSearch(problem: SearchProblem):一致代价搜索算法

aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):A\*搜索算法

**util.py:**包含了实现各种搜索算法可能用到的几种数据结构

class Stack:包含栈压入弹出判断为空的函数

class Queue:包含队列压入弹出判断为空的函数

class PriorityQueue:包含优先队列的压入弹出更新判断为空的函数

项目已有代码包括核心的search算法部分和用来解决不同问题的searchAgent部分，同时包括一些便于我们编写算法的工具，除了以上提及的部分之外，同时比较重要的部分还有problem类，类中包含了指向不同问题的接口，以及比较重要的getsuccessor方法和isgoalstate方法用来寻找后继搜索节点和判断是否到达目标，在编写算法前需要对这部分进行仔细阅读。

**3、子问题概述及简要分析**

**Q1：Finding a Fixed Food Dot using Depth First Search**

现在，pacman的世界只有wall和food，以及一个吃豆人，吃豆人需要从一个位置出发，到达指定goal的位置结束。

应用深度优先搜索，在给定目标坐标和当前坐标的情况下搜索出一条从当前位置到食物坐标处的路径，帮助吃豆人得到食物。深度优先搜索每次深入搜索树的底端，选一条路一直走到底，回溯，遍历所有的子节点，进而达到全局搜索的目的。

Q1需要实现一个在吃豆人地图中的dfs算法，填充search.py中的depthFirstSearch function。

**Q2：Breadth First Search**

应用广度优先搜索，在给定目标坐标和当前坐标的情况下搜索出一条从当前位置到食物坐标处的路径。广度优先搜索沿着搜索树的宽度进行遍历，逐层加深，从而遍历整颗搜索树。

Q2需要实现一个在吃豆人地图中的bfs算法，填充search.py中的depthFirstSearch function。

**Q3：Varying the Cost Function**

虽然BFS可以找到走向食物的最小步数，但是它不一定是最佳的。我们通过改变成本函数可以对Agent的行动进行一定程度上的控制，例如对充满食物的地方的每一步移动设置比较小的代价进行奖励，而对存在妖怪的危险地区的移动设置较高的代价进行惩罚，这样就可以使agent寻求一条比较安全且合适的路径寻找目标。

**Q4：A\* search**

不同于前面几种盲目搜索，A\*搜索是一种启发式搜索，通过设置启发式函数使Agent的行动更加只能，目标导向更加明确。A\*搜索每一步的代价是问题本身的代价加上启发式函数的值，在本项目的情景中，比较好用的启发式函数可能是曼哈顿距离或者欧拉距离，我们只需要在一致代价搜索的基础上将成本函数进行更改即可。

Q4需要实现一个在吃豆人地图中的A\*搜索算法，填充search.py中的aStarSearch function。

**Q5：Finding All the Corners**

此问题的目标是找到一个最短路径遍历地图的四个角落，我们采用A\*搜索，但是此次要自己编写移动的状态判断，如判断是否撞墙等。一旦访问一个角落，就把它标记为visited，并把访问过的角落数++，最后的截止状态是访问过的角落数等于4。

Q5需要我们重新定义problem，Q1~Q3的pacman都是到达一个具有food的特定位置，Q4中pacman需要到达map的4个角落：(1,1),(1,top),(right,1),(right,top）。我们需要编写searchAgents.py中的Class CornersProblem。

**Q6：Corners Problem: Heuristic**

此问题的目标是找到一个最短路径遍历地图的四个角落，我们同样采用A\*搜索，不过这次要自己设置启发式函数，要根据具体的问题设置比较一致和完备的启发式函数，就可以显著降低搜索的成本和拓展的节点数。

Q6需要为Q5定义的problem设计一个启发式函数。

**Q7：Eating All The Dots**

Q7需要为一个新的problem（FoodSearchProblem）设计启发式函数，该problem定义与上述不同的地方在于，map中有多个坐标点上存在food，pacman需要将food吃完，达到目的。problem定义状态为元祖(position, foodGrid)，第一个参数为pacman所在坐标，第二个参数为pacman所在map大小的网格（矩阵)，矩阵每一个元素为true or false，true表示该位置上还有food，false表示该位置没有food。于是goal state即为state中参数二全为false。

**Q8：Suboptimal Search**

Q8需要为一个ClosestDotSearchAgent寻找到次优解，agent会指导pacman在一张具有多个food的map中行走，指导方法是pacman每次先去到最近的food处，然后再去下一个最近的food处，直到把所有food吃完。

1. 算法设计与实现

**Q1：Finding a Fixed Food Dot using Depth First Search**

分析发现深度优先搜索的顺序和栈的压入弹出的顺序之间存在着某种联系，因此采用栈的压入弹出来实现深度优先搜索。所以对于DFS问题，我们采用栈这一数据结构，建立两个表，一个为stack表，初始化为栈，用来存放当前没有被拓展到的节点；另一个是visited表，用来存放已经访问过的节点。

每一次搜索的步骤如下：

1. 把初始节点S放入stack表

2. 如果stack表为空，那么问题无解，退出

3. 从stack栈中弹出一个节点p，访问并放入visited表

4. 考察节点p是否是目标节点，如果是则退出，否则继续搜索

5. 若节点p不可拓展，回到第2步，

6. 否则拓展节点p，将其子节点压栈，然后再重复2-6步操作，直到求得问题的解。

def depthFirstSearch(problem: SearchProblem):

from util import Stack

    from game import Directions

    # 初始化栈，用于存放待探索的节点

    stack = util.Stack()

    # 初始化集合，用于记录已访问过的节点

    visited = set()

    # 将初始节点以及空的动作序列压入栈中

    stack.push((problem.getStartState(), []))

    while not stack.isEmpty():

        # 弹出栈顶节点及其对应的动作序列

        state, actions = stack.pop()

        # 如果当前节点是目标节点，返回到达目标节点的动作序列

        if problem.isGoalState(state):

            return actions

        # 如果当前节点未被访问过

        if state not in visited:

            # 将当前节点标记为已访问

            visited.add(state)

            # 获取当前节点的所有子节点

            successors = problem.getSuccessors(state)

            for next\_state, action, cost in successors:

                # 如果子节点未被访问过，将其及对应动作序列压入栈中

                if next\_state not in visited:

                    stack.push((next\_state, actions + [action]))

    # 若栈为空且未找到目标节点，则返回空列表表示无解

    return []

深度优先搜索算法的优点是比较简单、容易实现，只要储存当前路径，就比较有希望在更浅处找到问题的解，求解速度相对较快。但是一旦有环，可能会陷入死循环，并且如果不限制深度的话可能会导致大量的资源浪费，并且找到的可能不是最优解，还可能会出现栈溢出问题。如果目标节点更靠近叶子节点，DFS会更好用，对于不要求最优解的情况，DFS内存消耗少，速度较快，较为适用。

**Q2：Breadth First Search**

分析发现广度优先搜索的顺序和队列的入队和出队顺序之间存在着一定联系，因此采用队列的出队入队来实现广度优先搜索。所以对于BFS问题，我们采用的是队列这一数据结构，建立两个表，一个为queue表，初始化为队列，存放当前没有拓展的节点；另一个是visited表，用来存放已经访问过的节点。

每一次搜索的步骤如下：

1.把初始节点S放入queue表

2.如果queue表为空，那么问题无解，退出

3.一个节点p从queue队列中出队，访问并放入visited表

4．考察节点p是否是目标节点，如果是则退出，否则继续搜索

5．若节点p不可拓展，回到第2步，

6.否则拓展节点p，将其子节点入队，然后再重复2-6步操作，直到求得问题的解。

def breadthFirstSearch(problem: SearchProblem):

    # 初始化队列，用于存放待探索的节点

    queue = util.Queue()

    # 初始化集合，用于记录已访问过的节点

    visited = set()

      # 将初始节点以及空的动作序列入队

    queue.push((problem.getStartState(), []))

    while not queue.isEmpty():

        # 出队一个节点及其对应的动作序列

        state, actions = queue.pop()

        # 如果当前节点是目标节点，返回到达目标节点的动作序列

        if problem.isGoalState(state):

            return actions

        # 如果当前节点未被访问过

        if state not in visited:

            # 将当前节点标记为已访问

            visited.add(state)

            # 获取当前节点的所有子节点

            successors = problem.getSuccessors(state)

            for next\_state, action, cost in successors:

                # 如果子节点未被访问过，将其及对应动作序列入队

                if next\_state not in visited:

                    queue.push((next\_state, actions + [action]))

    # 若队列为空且未找到目标节点，则返回空列表表示无解

    return []

广度优先搜索从距离起点最近的节点开始搜索，保证找到最短路径，它按照层次顺序搜索，所以可以找到最优解。但是需要维护一个队列来存储每个节点的邻居节点，所以空间复杂度较高，当目标图像非常大时，BFS可能无法完成搜索。

**Q3：Varying the Cost Function**

这个问题本质上就是广度优先搜索加上最低成本的思想，广度优先搜索每一步的成本为1，而这个搜索就是通过改变成本函数鼓励吃豆人找到离食物更近的那条路。

为此我们只需要把queue表设置成为一个优先队列，进队之后自动按当前代价排序，下一个出队的一定是当前所有拓展节点中成本最小的那个。这个思想的关键在于调用函数getcostofactions，获得每一步的最小成本。

def uniformCostSearch(problem: SearchProblem):

    # 初始化集合，用于记录已访问过的节点

    visited = set()

    # 初始化优先队列，用于存放待探索的节点，优先级为节点的代价

    queue = util.PriorityQueue()

    # 将初始节点以及空的动作序列和代价0入队,其元素优先级为0

    queue.push((problem.getStartState(), [], 0), 0)

    while not queue.isEmpty():

        # 从优先队列中出队一个节点及其对应的动作序列和代价

        state, actions, cost = queue.pop()

        # 如果当前节点是目标节点，返回到达目标节点的动作序列

        if problem.isGoalState(state):

            return actions

        if state not in visited:

            # 将当前节点标记为已访问

            visited.add(state)

            # 获取当前节点的所有子节点

            successors = problem.getSuccessors(state)

            for next\_state, action, step\_cost in successors:

                new\_cost = cost + step\_cost

                # 如果子节点未被访问过，或者新的代价更低，则更新优先队列

                if next\_state not in visited:

                    queue.push((next\_state, actions + [action], new\_cost), new\_cost)

                # 如果新的代价更低，则更新优先队列

                elif next\_state in visited and new\_cost < cost:

                    queue.update((next\_state, actions + [action], new\_cost), new\_cost)

    # 若队列为空且未找到目标节点，则返回空列表表示无解

    return []

这个搜索算法在广度优先搜索的基础上考虑了成本，用改变成本函数来促使吃豆人改进他的行为，并且一旦检测到更优的新路径，就会丢弃老路径。但是如果存在零成本时就会出现死循环，而且这个搜索算法具有较高的时间复杂度，因为要想搜索到成本小的步骤，必须搜索很大的搜索树，找到所有节点确定其为最小成本时才会终止。这个搜索算法适用于每一步成本不等的情况下，亦或是需要对搜索行为进行一定控制的情况，而不太适用于存在零成本行为的问题。

**Q4：A\* search**

A\*算法是一种启发式算法，其每一步的成本不仅包括实际上的成本，还包括一个启发式函数的值，在方格地图中，通常把启发式函数设置为当前节点到终点的曼哈顿距离。

A\*搜索是将启发式函数值和路径成本的和作为扩展的依据，因此将Q3问题中的搜索算法中的路径成本改为路径成本和启发函数的值的和来进行优先队列的入队操作就能实现A\*搜索算法。

def aStarSearch(problem: SearchProblem, heuristic=nullHeuristic):

    # 初始化集合，用于记录已访问过的节点

    visited = set()

    # 初始化优先队列，用于存放待探索的节点，优先级为节点的代价加上启发函数值

    queue = util.PriorityQueue()

    # 将初始节点以及空的动作序列和代价0入队，优先级为启发函数值

    queue.push((problem.getStartState(), [], 0), heuristic(problem.getStartState(), problem))

    while not queue.isEmpty():

        # 从优先队列中出队一个节点及其对应的动作序列和代价

        state, actions, cost = queue.pop()

        # 如果当前节点是目标节点，返回到达目标节点的动作序列

        if problem.isGoalState(state):

            return actions

        # 如果当前节点还未访问过

        if state not in visited:

            # 将当前节点标记为已访问

            visited.add(state)

            # 获取当前节点的所有子节点

            successors = problem.getSuccessors(state)

            for next\_state, action, step\_cost in successors:

                new\_cost = cost + step\_cost

                priority = new\_cost + heuristic(next\_state, problem)

                # 如果子节点未被访问过，或者新的代价更低，则更新优先队列

                if next\_state not in visited:

                    queue.push((next\_state, actions + [action], new\_cost), priority)

                # 如果新的代价更低，则更新优先队列

                elif next\_state in visited and new\_cost < cost:

                    queue.update((next\_state, actions + [action], new\_cost), priority)

    # 若队列为空且未找到目标节点，则返回空列表表示无解

    return []

A\*算法是目前人们解决实际问题中用的较多的方法，A\*算法的效率和优势主要取决于其启发式函数的设计，如果启发式函数设计的越符合问题的实际情况，那么搜索的效率就会越高。并且在给定一致的启发式函数的情况下，A\*算法一定是效率最优的。但是当状态空间中存在多个目标状态时，或者是比较接近最佳目标状态时，搜索有可能因为启发式函数而误入歧途，带来不小的额外成本，而由于较大的时间复杂度，在遇见复杂问题时，用A\*算法可能难以找到一条最佳路径，而是只能满足局部最优解；同时由于A\*算法在内存中保留了所有已经生成的节点，所以在面对大规模的搜索问题时，A\*算法很容易耗尽内存。

**Q5：Finding All the Corners**

这一部分要求遍历所有角落，要补全getStartState，isGoalState，getSuccessors等几个函数，本题要设置一个合适的状态来记录。因为在遍历四个顶点的过程中，不可避免的会出现前后经过相同位置的情况，因此如果再将位置作为状态空间，则不好处理是否已经扩展的问题，另外由于游戏是否完成还取决于四个角落是否被经过，因此考虑将四个角落是否被经历过作为状态的一部分，既利于判断是否到达结束状态，又巧妙的避开了状态空间重复的问题。之后利用宽度优先搜索就能完成遍历四个顶点的目标了。

    def getStartState(self):

        # 初始状态为吃豆人的起始位置和空的已访问角落集合

        return (self.startingPosition, ())

    def isGoalState(self, state: Any):

        # 检查当前状态是否已经访问了所有角落

        return len(state[1]) == 4

    def getSuccessors(self, state: Any):

        successors = []

        for action in [Directions.NORTH, Directions.SOUTH, Directions.EAST, Directions.WEST]:

            x, y = currentPosition = state[0]  # 当前位置

            dx, dy = Actions.directionToVector(action)  # 获取动作对应的方向向量

            nextx, nexty = int(x + dx), int(y + dy)  # 计算下一个位置

            hitsWall = self.walls[nextx][nexty]  # 检查下一个位置是否是墙壁

            if not hitsWall:

                # 如果下一个位置不是墙壁，则添加一个后继状态到后继列表中

                visitedCorners = state[1]  # 已访问的角落集合

                next = (nextx, nexty)  # 下一个位置

                if next in self.corners and next not in visitedCorners:

                # 如果下一个位置是一个未访问的角落，则更新已访问的角落集合

                    visitedCorners = visitedCorners + (next,)

                nextState = (next, visitedCorners)  # 下一个状态

                successors.append((nextState, action, 1))  # 添加后继状态到后继列表中，成本为1

        self.\_expanded += 1 # DO NOT CHANGE

        return successors

**Q6：Corners Problem: Heuristic**

考虑到要设计一个效率较高的算法找到一条相对好的路径，本题要求自己设计启发式函数来提高搜索效率，降低搜索拓展的节点数。关于启发式函数，在宽度优先搜索时就进行过启发式函数的扩展，因此这里模仿那道题的思路，采用四个顶点中曼哈顿距离最大的作为启发式函数的值。

def min\_distance(corners, current\_pos):

    #选择所有未访问角落中的最小曼哈顿距离作为启发式函数。

    if len(corners) == 0:

        return 0

    distances = []

    for corner in corners:

        distance = abs(corner[0] - current\_pos[0]) + abs(corner[1] - current\_pos[1])

        distances.append(distance + min\_distance([c for c in corners if c != corner], corner))

    return min(distances)

def cornersHeuristic(state: Any, problem: CornersProblem):

    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 \*\*\*"

   # 获取当前位置和已访问的角落集合

    current\_pos, visited\_corners = state

    # 计算所有未访问角落的最小曼哈顿距离作为启发式值

    unvisited\_corners = [corner for corner in corners if corner not in visited\_corners]

    heuristic\_value = min\_distance(unvisited\_corners, current\_pos)

    return heuristic\_value

**Q7：Eating All The Dots**

此问题要设计一个启发式函数使Agent遍历所有食物并使拓展的节点尽可能少。由于是要找到地图中所有食物，所以可以把算法的启发式函数用一个已有的函数Mazedistance来求解，由于要满足一致性，即到终点的代价要小于现在状态到中间状态和中间状态到最终状态的和，所以必须选所有地图距离中最大的一个值作为启发函数，当吃豆人到所有食物的最大距离变小时，代表其越接近完成目标，他就会一直朝着这个方向前进，这样可以显著减少拓展的节点。

def foodHeuristic(state: Tuple[Tuple, List[List]], problem: FoodSearchProblem):

    position, foodGrid = state

    "\*\*\* YOUR CODE HERE \*\*\*"

    # 计数食物数量

    count = 0

    # 复制食物网格以免修改原始数据

    food = foodGrid.copy()

    # 获取Pacman的位置

    location = position

    # 初始化最大距离和最小距离

    max\_distance = 0

    max\_location = [0, 0]

    min\_location = [0, 0]

    min\_distance = 100000

    # 获取食物网格的高度和宽度

    height = food.height

    width = food.width

    # 遍历食物网格

    for i in range(width):

        for j in range(height):

            # 如果当前位置有食物

            if food[i][j]:

                # 增加食物计数

                count += 1

                # 计算当前食物到Pacman的曼哈顿距离

                distance = abs(i - location[0]) + abs(j - location[1])

                # 更新最大距离和对应的食物位置

                if distance > max\_distance:

                    max\_distance = distance

                    max\_location = [i, j]

                # 更新最小距离和对应的食物位置

                if distance < min\_distance:

                    min\_distance = distance

                    min\_location = [i, j]

    # 如果没有食物，则返回0

    if count == 0:

        return 0

    # 如果只有一个食物，则返回当前位置到最远食物的距离

    elif count == 1:

        return abs(max\_location[0] - location[0]) + abs(max\_location[1] - location[1])

    # 如果有多个食物，则返回当前位置到最近和最远食物的距离之和

    else:

        return abs(min\_location[0] - location[0]) + abs(min\_location[1] - location[1]) + \

               abs(min\_location[0] - max\_location[0]) + abs(min\_location[1] - max\_location[1])

**Q8：Suboptimal Search**

由于A\*算法不一定每次都能返回一个最佳路径，我们有时候可以退而求其次，本题要求控制吃豆人每次都吃最近的食物，而这种情况下需要补全一个找到最近食物的函数和一个判断结束状态的函数。本问题的结束状态可以是找到当前节点到所有节点的曼哈顿距离，然后找出其中最小的那个，如果最小的那个为0，说明此次搜索实现了找到当前最近的节点。而对于找路径的算法，我选择的是之前已经编写好的A\*算法，于是只需要补充findPathToClosestDot的返回值即可。

    def findPathToClosestDot(self, gameState: pacman.GameState):

        # Here are some useful elements of the startState

        startPosition = gameState.getPacmanPosition()

        food = gameState.getFood()

        walls = gameState.getWalls()

        problem = AnyFoodSearchProblem(gameState)

        "\*\*\* YOUR CODE HERE \*\*\*"

        # 获取开始状态的一些有用元素

        return search.aStarSearch(problem)

    def isGoalState(self, state: Tuple[int, int]):

        x,y = state

        "\*\*\* YOUR CODE HERE \*\*\*"

        # 获取食物位置列表

        foodLocations = self.food.asList()

        # 计算当前位置到所有食物位置的曼哈顿距离

        distances = []

        for food in foodLocations:

            distances.append(util.manhattanDistance(state, food))

        # 获取最小距离

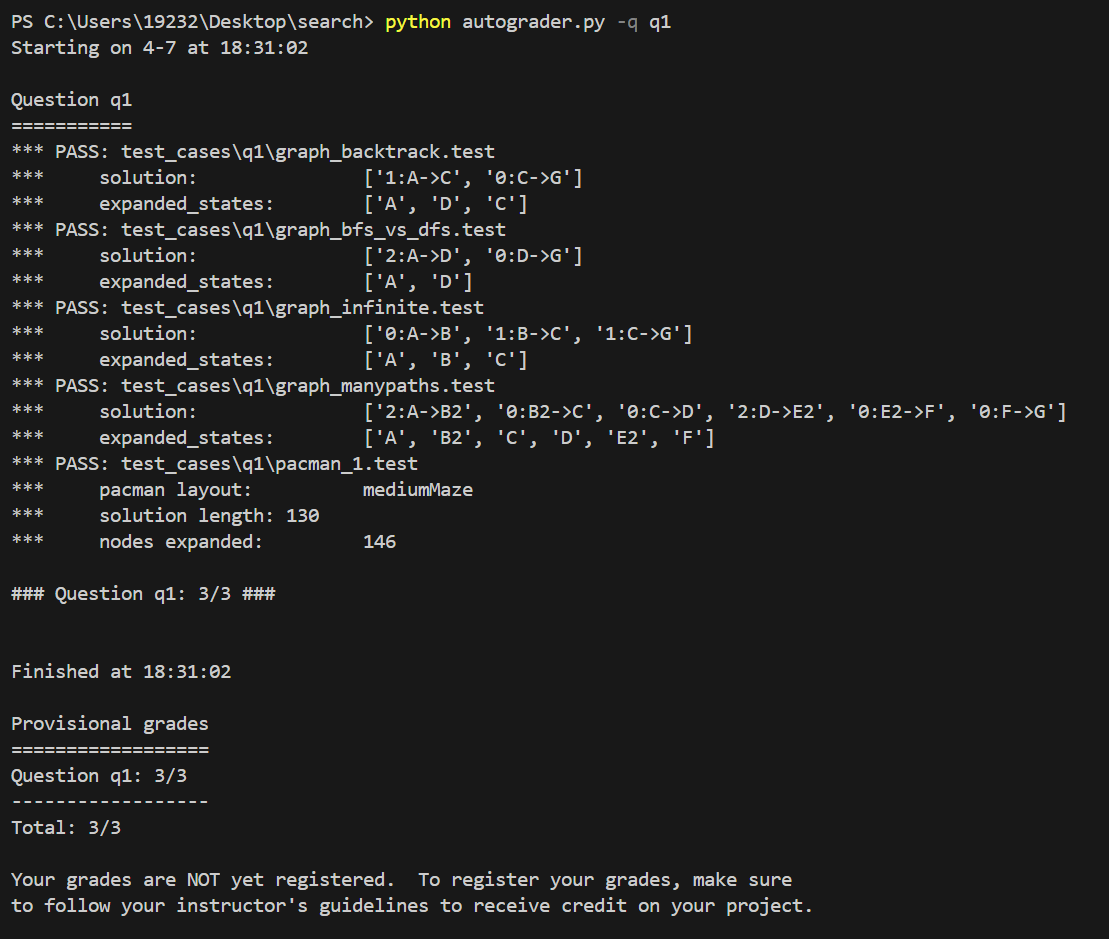
        minDistance = min(distances)

        # 判断最小距离是否为0，即当前位置是否为食物位置

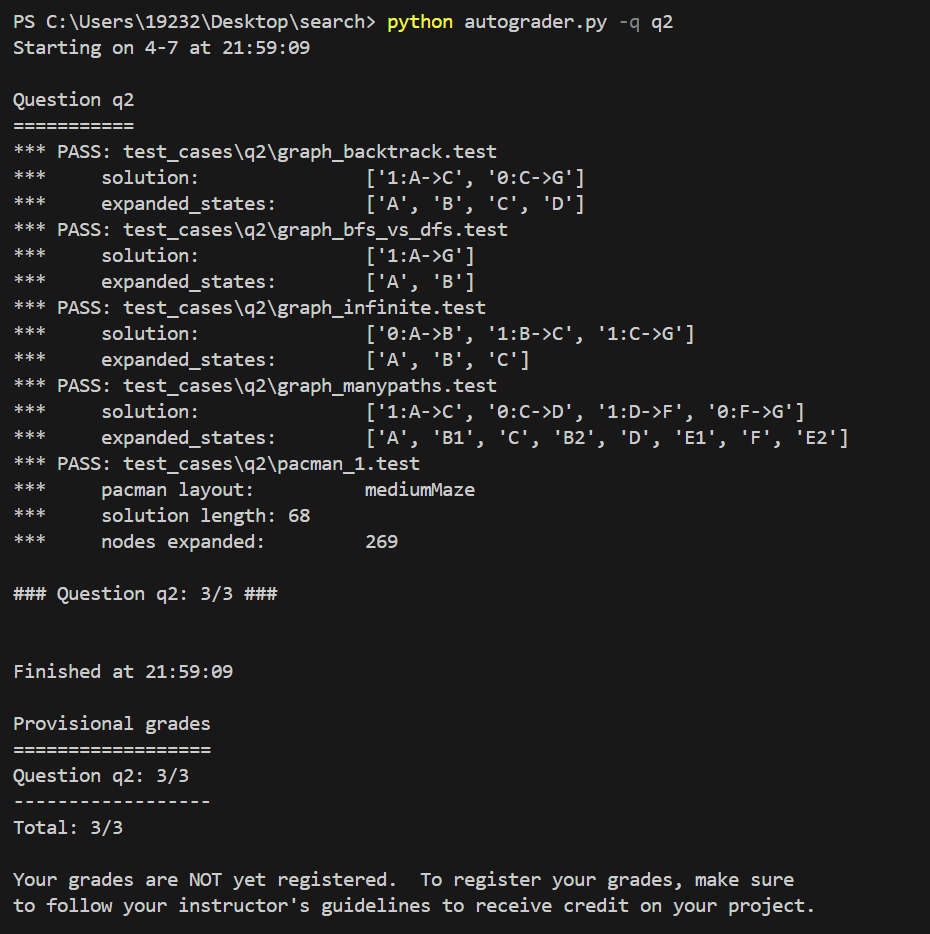
        return minDistance == 0

1. 结果展示

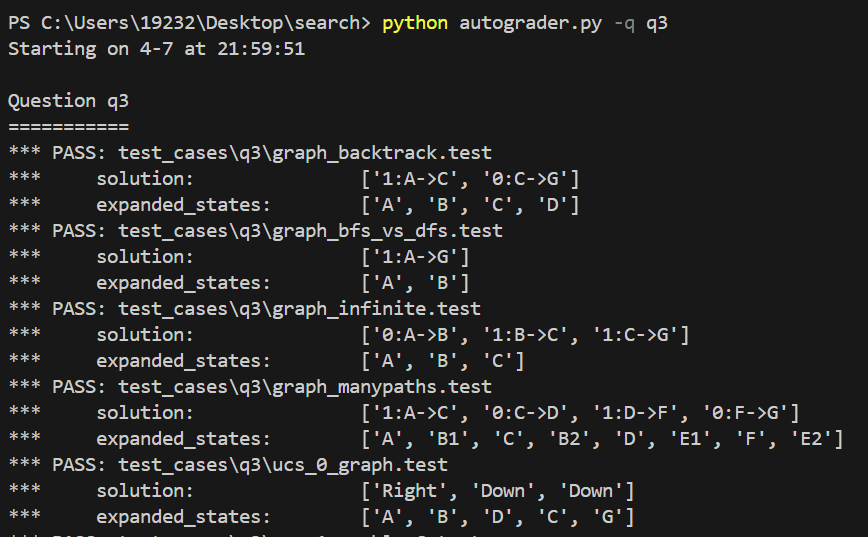
**Q1：Finding a Fixed Food Dot using Depth First Search**

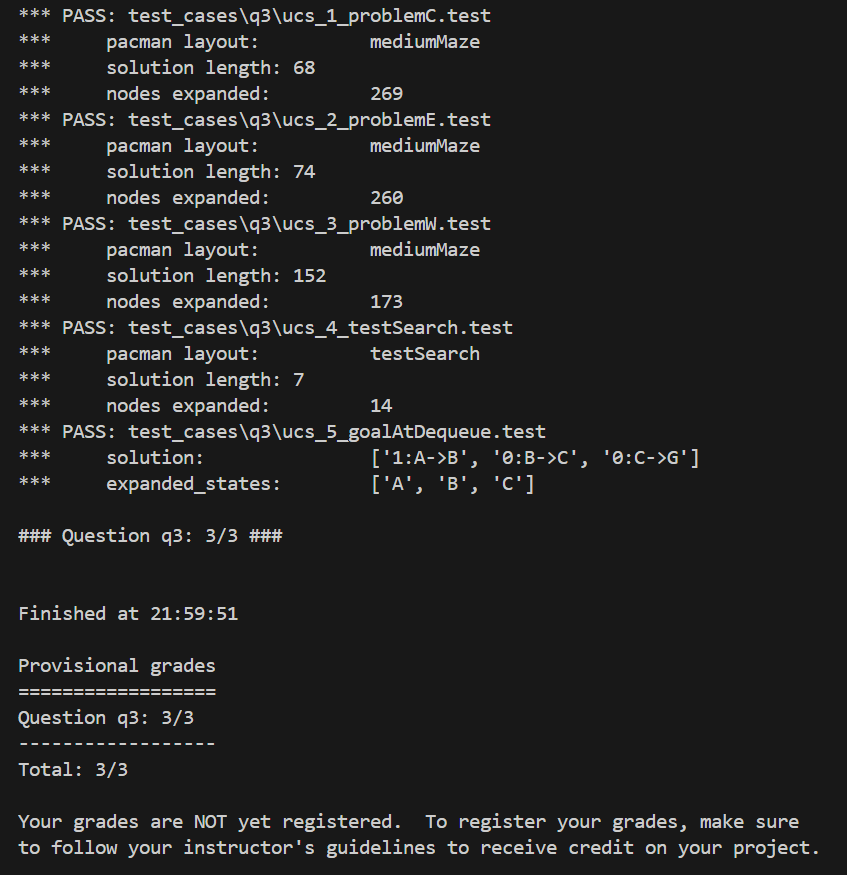
****

**Q2：Breadth First Search**

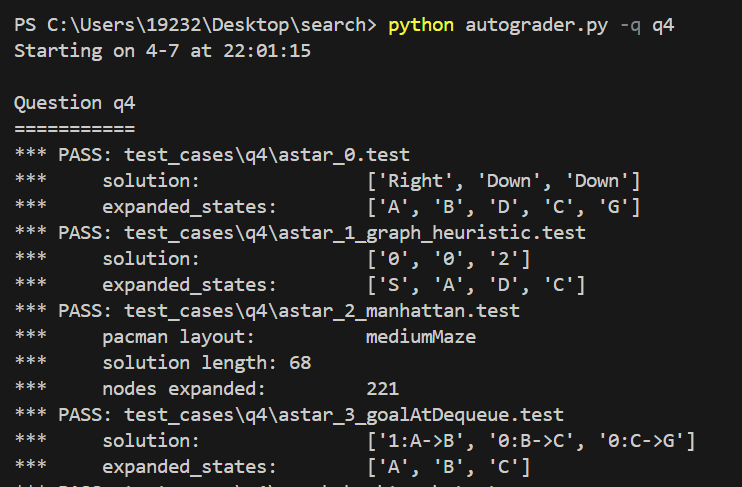


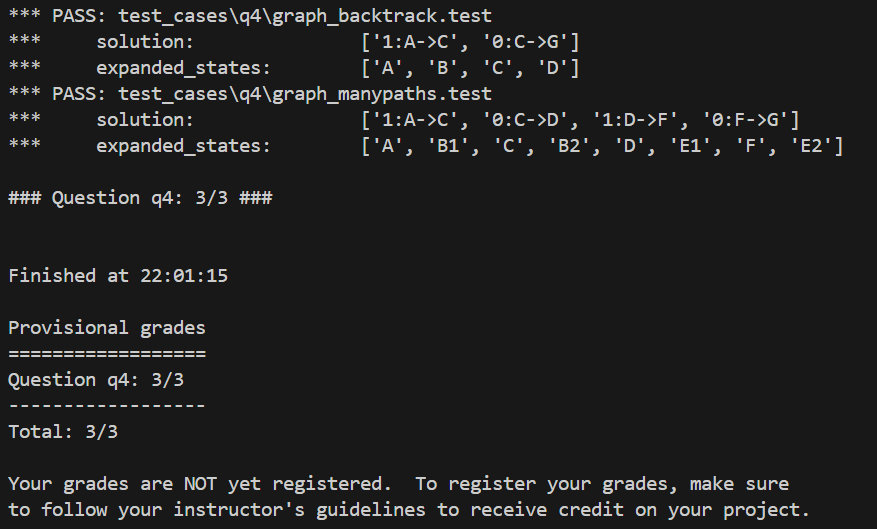
**Q3：Varying the Cost Function**

****

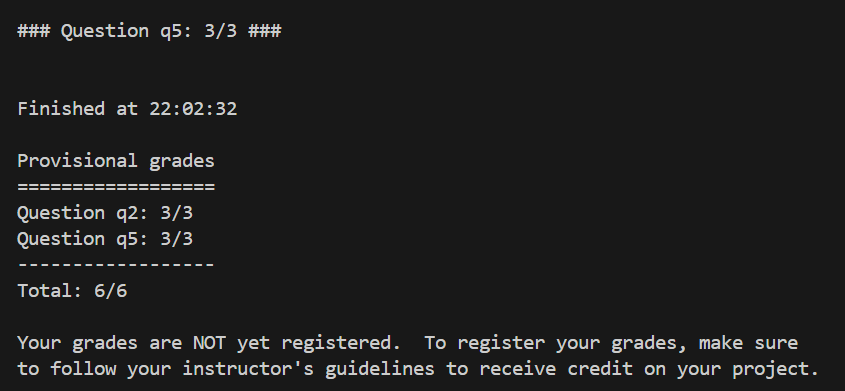
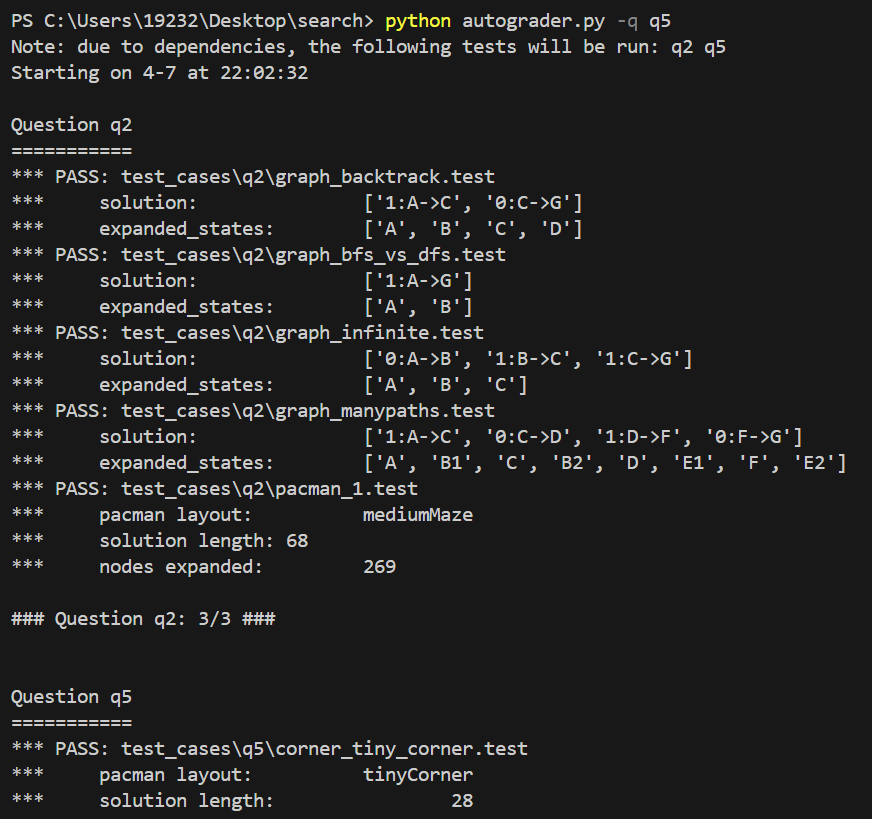


**Q4：A\* search**

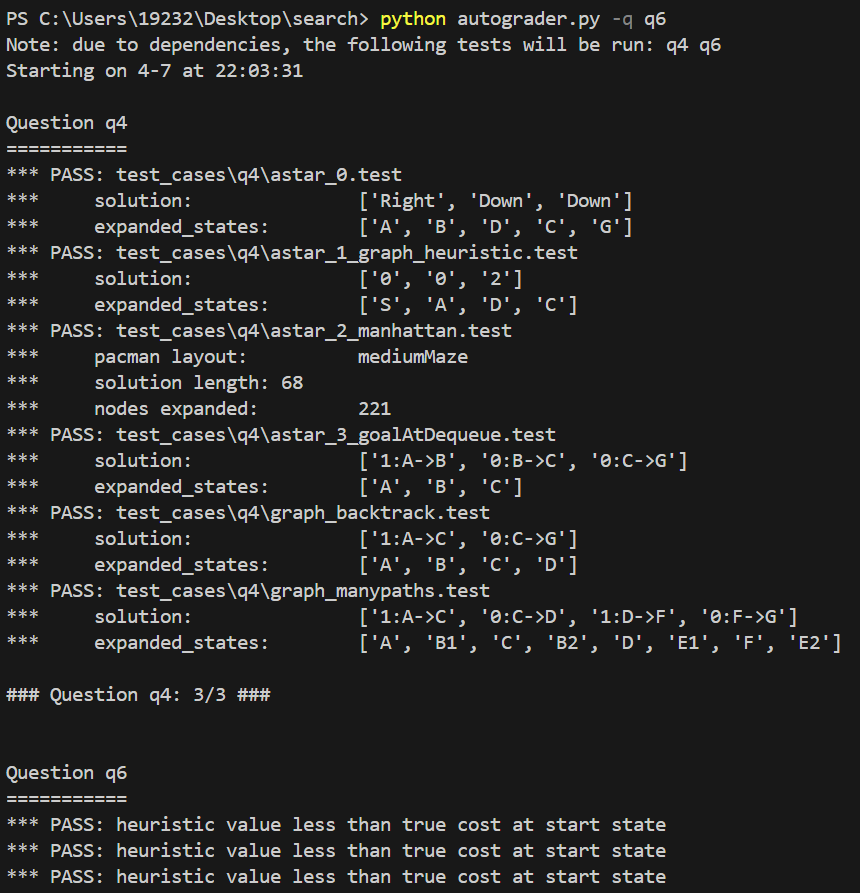
****

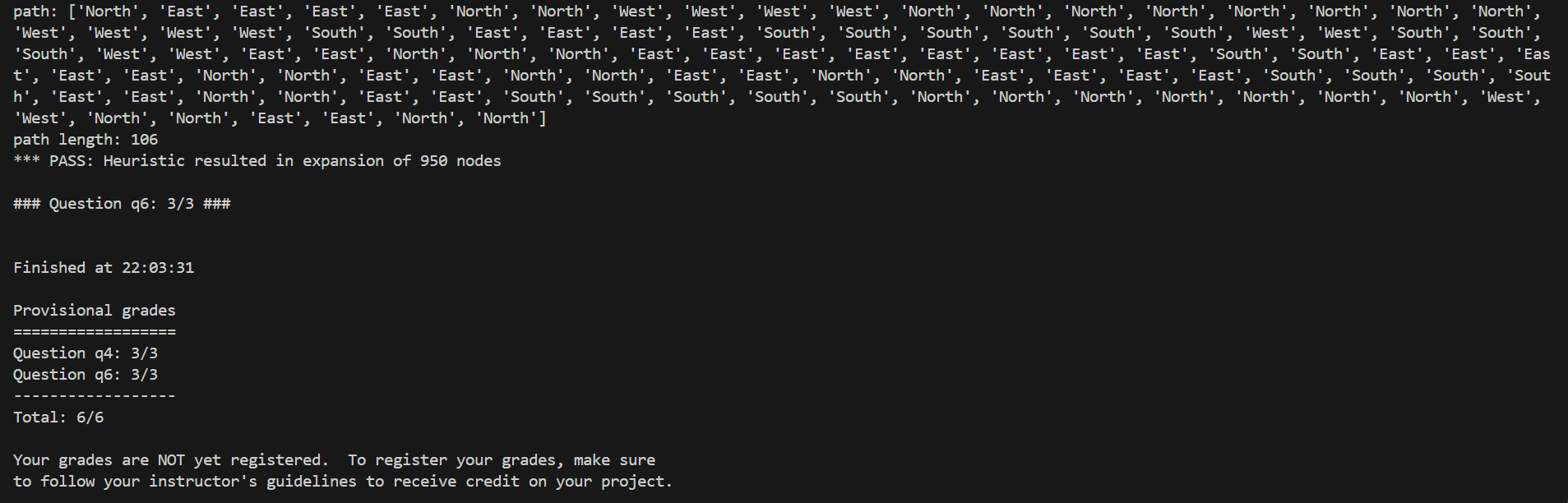


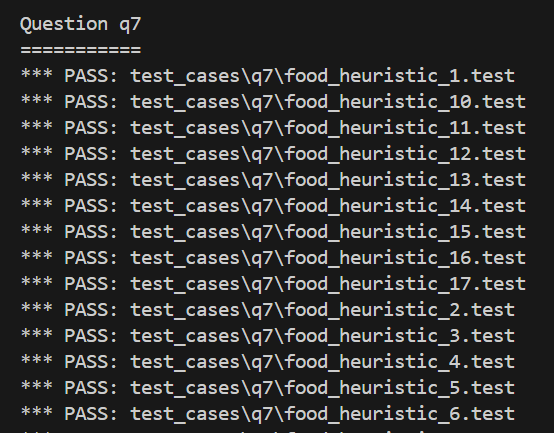
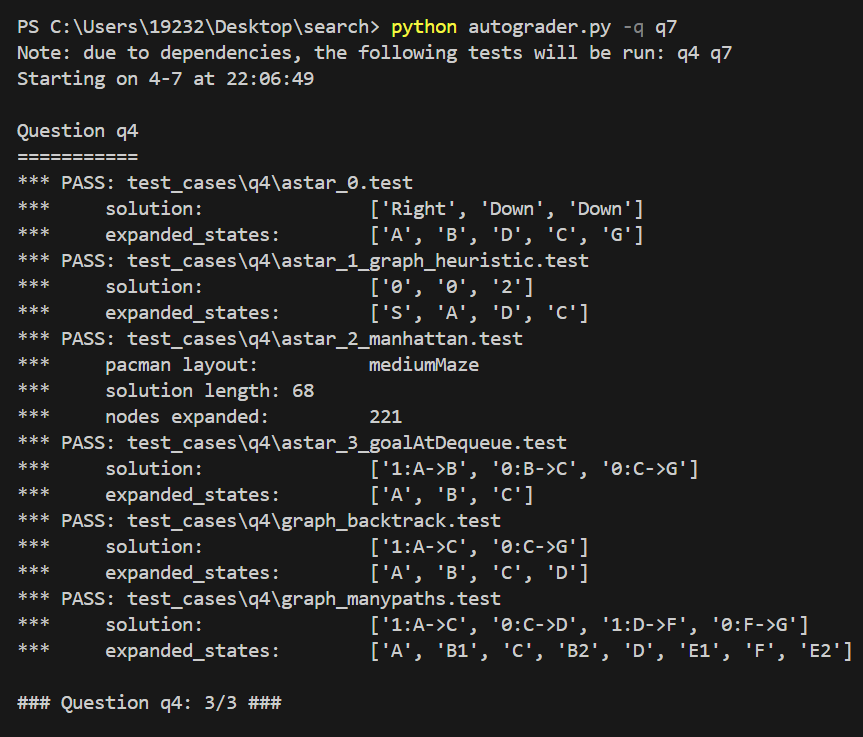
**Q5：Finding All the Corners**

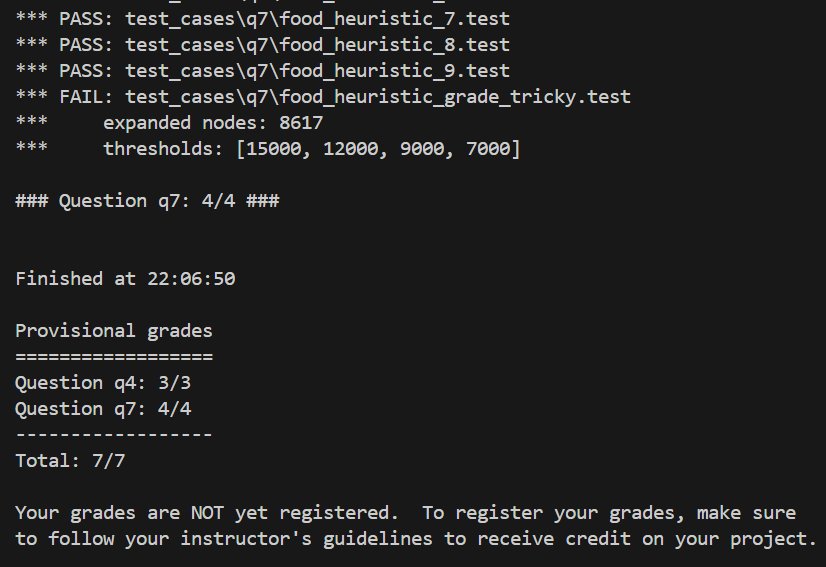


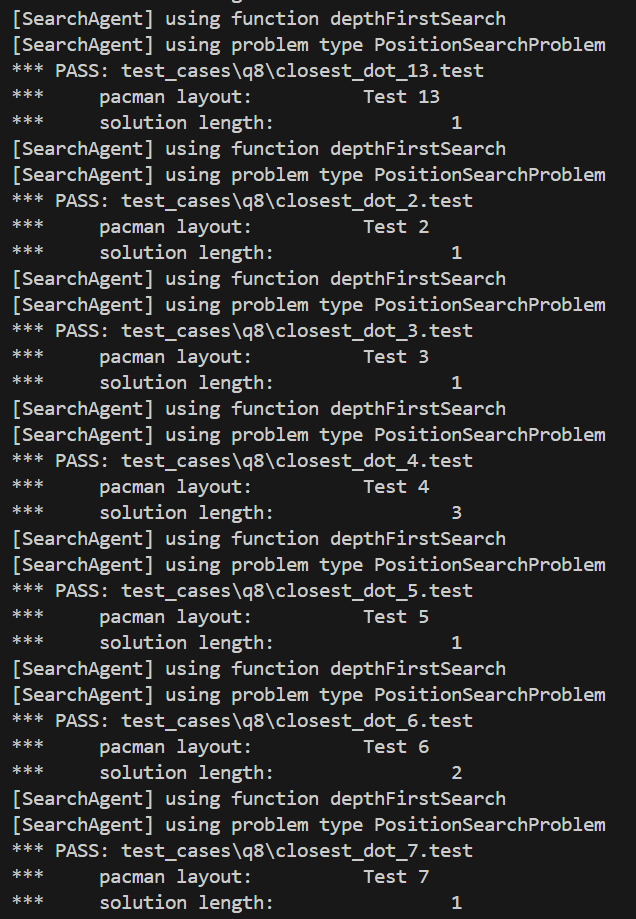
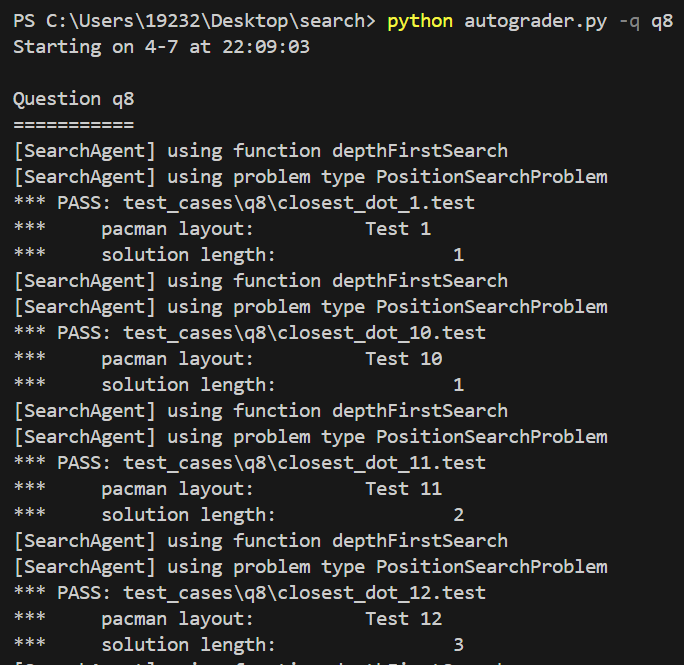
**Q6：Corners Problem: Heuristic**

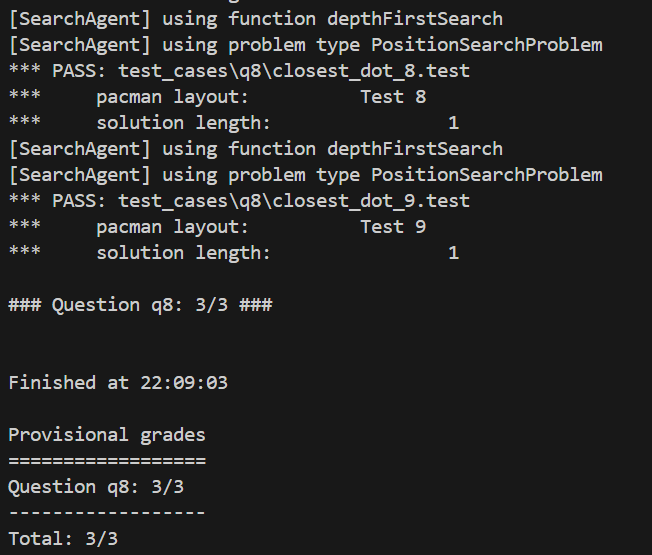


**Q7：Eating All The Dots**

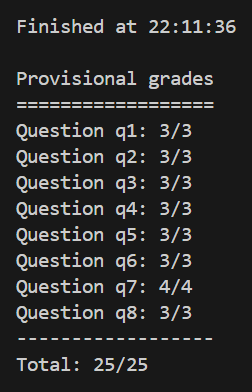


**Q8：Suboptimal Search**





**Total Result**



1. 总结与分析

**1、算法分析**

在解决寻路问题时，选择合适的搜索算法非常重要。深度优先搜索和宽度优先搜索是最基本的搜索算法，它们分别具有深度优先和广度优先的特点。A\*搜索是更高级的搜索算法，它可以使用启发式函数来指导搜索方向。在实际应用中，要根据具体问题的特点来选择合适的搜索算法。

启发式函数可以提供搜索方向的指引，对搜索效率有重要的影响。设计好的启发式函数应该尽可能地接近实际问题的情况，可以根据问题的不同特点来设计不同的启发式函数。

在实现算法时，需要注意算法的细节。例如，在A\*搜索中，需要给每个状态赋一个成本值，并在更新成本值时进行优化。还需要注意避免重复搜索同一个状态，以及避免搜索无效的状态。

不同的搜索算法具有不同的优缺点。例如，深度优先搜索和宽度优先搜索在空间和时间上的复杂度差异很大， A\*搜索需要选择合适的启发式函数来达到最优解。

**2、项目总结**

**2.1 遇到的困难及解决方法：**

在完成此项目过程中，在代码方面遇到了一些困难，主要还是因为没有掌握python语言导致的，例如关于problem类的继承我一开始就不理解。这就导致我在项目刚开始时看已有代码的过程中用掉了大量时间，不过也得到了相应的成效，花费大量时间把项目要求仔细阅读的代码逻辑理顺后，再补全算法函数就比较简单了。还有就是本题中的搜索算法涉及到了一些复杂的概念，比如图搜索、启发式搜索等，这些概念可能对我来说有些晦涩难懂，我也花费了一些时间去了解相关的概念。

在实现具体的搜索算法时，我也遇到了一些细节上的小问题，比如数据结构的选择、如何表示状态和操作等，我参考了网站上提供的代码，努力理解其中的细节。对于我来说，最难的应该就是调试了，在编写代码时难免会出现一些错误，特别是在实现较复杂的算法时，调试代码算是一个很大的挑战。

**2.2 项目总结与个人收获：**

通过这次项目，我算是比较好地完成了项目要求任务，由于对python基础语法不够熟悉，项目实现过程中遇到了不少困难，但是经过仔细思考和反复斟酌，最终还是能够克服这些困难。经过8个问题的循序渐进，我不但增长了我对人工智能搜索问题的理解，而且夯实了我的python程序设计基础，这是一次非常有意义的实践。