Task2

实验报告 20214809 凌鹏

Question 1 (4 points): Finding a Fixed Food Dot using Depth First Search：

实验记录：在完成 DFS 算法之前，我们需要先对整个迷宫的数据结构进行了解，主要包括：

①迷宫的表示形式。在查看 **search.py** 和 **game.py** 等源码后，发现 *"problem.walls"* 代表当前的迷宫形状，其类型为 **Grid**，定义在 **game.py** 中，并且每一个位置的值都是 bool 类型，True 表示为墙壁，False 表示为道路。

②起始点和终止点位置以及表达形式。查看源码后，发现起始点表示为"***problem.getStartState()***"，而终止点表示为"***problem.goal***"，每一个状态都是一个 **(i, j)** 元组，可以通过"***problem. isGoalState(nowState)***"来判断当前是否已经到达终点。

③迷宫中"行走"的方式，主要是指行走的方向。经过查看源码，发现 **game.py** 文件中定义了 ***Actions*** 和 ***Directions*** 两个相关类，动作形式也同时定义。

④我们发现不同的 **problem** 类对于具有不同的属性和方法（比如 ***searchAgents.PositionSearchProblem*** 类就不具备 **getExpandedStates()方法，而 searchTestClasses.GraphSearch 类就有该方法)**，这种差异要求我们在编写 DFS 函数时必须小心的处理。

⑤使用 ***autograde.py*** 函数进行评价时我们发现，正确答案包括两个部分，一个是路径（类似 A->B 这种），此时最后一条路径的终点必定是**目标**位置；另一部分是状态转变（类似[A, B, C]这种），其实就是每条路径的起始点，所以一定不能包含目标位置在内。这一差异又要求我们在 DFS 中特殊处理。

了解上述的基本情况后，就可以动手实现 DFS 算法，思路比较简单，不做过多叙述，唯一需要注意的：不要忘记记录已经走过的位置。下面是结果截图：

```
Question q1
===========
<class 'searchTestClasses.GraphSearch'>
False
True
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
<class 'searchTestClasses.GraphSearch'>
False
True
*** PASS: test_cases\q1\graph_bfs_vs_dfs.test
***     solution:               ['0:A->B', '0:B->D', '0:D->G']
***     expanded_states:        ['A', 'B', 'D']
<class 'searchTestClasses.GraphSearch'>
False
True
*** PASS: test_cases\q1\graph_infinite.test
***     solution:               ['0:A->B', '1:B->C', '1:C->G']
***     expanded_states:        ['A', 'B', 'C']
<class 'searchTestClasses.GraphSearch'>
False
True
*** 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']
<class 'searchAgents.PositionSearchProblem'>
True
True
*** PASS: test_cases\q1\pacman_1.test
***     pacman layout:          mediumMaze
***     solution length: 246
***     nodes expanded:         269

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

图 1：Question1 结果图

Question 2 (4 points): Breadth First Search：

实验记录：这个实验在完成的时候问题比较多。首先，主要是由于不同的 problem 类函数和属性存在差异，导致我们需要在 BFS 内部实现兼容。其次，由于最终需要的是从起点到终点的路径，由于 BFS 的特征，在保存每一次 BFS 的状态转移时（包括动作），必须保存相反的状态转移，以便最终根据目标反向回溯。Autograde 结果如下图所示；总的来说 BFS 比 DFS 在该项任务中更难。

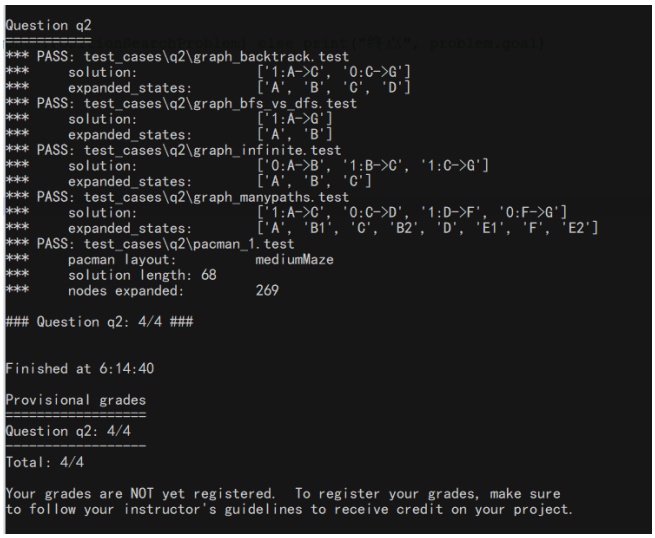

图 2：Question2 结果图

Question 3 (4 points): A* search：

实验记录：A*算法总体思路和 BFS 类似，但是与 BFS 不同点在于：BFS 每个状态的优先级一样，如果使用队列装载所有的状态，那么每个状态出队时按照先进先出的规则即可；而如果使用的是 A*算法，那么从状态队列中取出下一个转移状态之前，需要考虑队列中所有状态的价值（cost），在 Q3 中则是直接使用的城市距离来衡量每个状态的 cost：计算每个状态与终点 goal 直接的城市距离。因此，在实现的时候，我们只需要将队列改为优先队列，然后操作方式和 BFS 一样即可。实验结果如:3 所示。

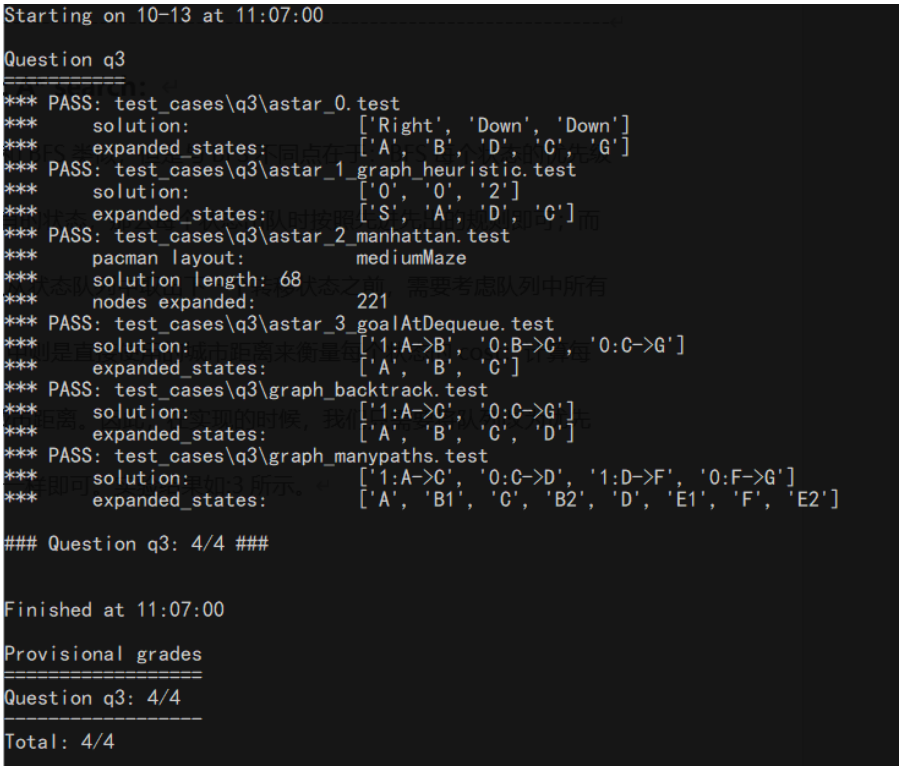

图 3：Question3 结果图

Question 4 (3 points): Finding All the Corners

实验记录：比较简单，直接上图 4。

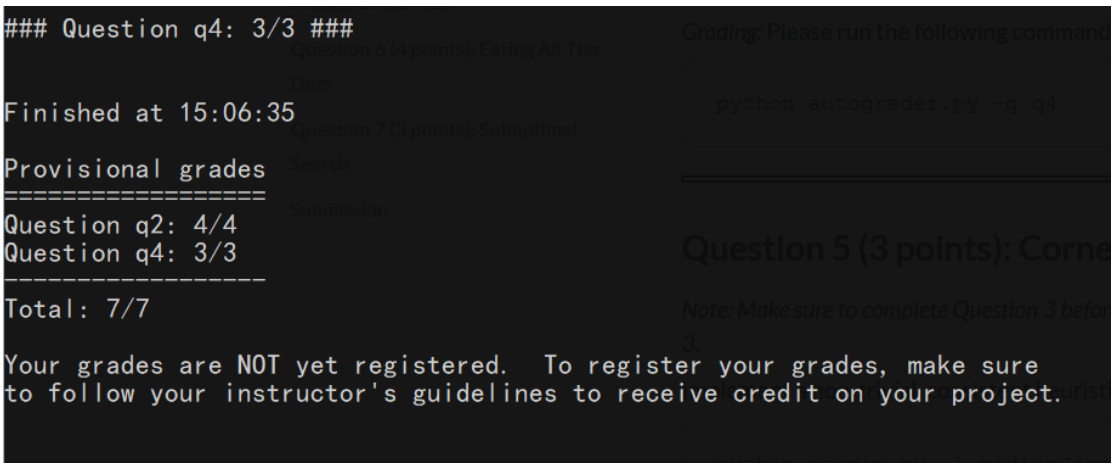

图 4：Question4 结果图

Question 5 (3 points): Corners Problem: Heuristic

实验记录：Q5 需要我们设计一个能够使得 agent 从起始点出发，使用最短时间到达四个

边角的启发式算法。在这之前，我们完成过从起始点使用最短时间到达一个终点的方法，在那里我们使用了 A*算法来完成。然后我们回到现在的问题，并且思考得到解决方案如下：1）找到当前状态时，所有的尚未访问过的终点 corners；2）我们计算从当前状态(state)到所有未访问过的终点之间的距离，这里使用 DFS 计算最佳；3）我们选择返回从当前状态到所有未访问过的 corners 的距离最大的那个作为 cost。通过上述三步即可在尽可能短的时间内访问所有的 corners。

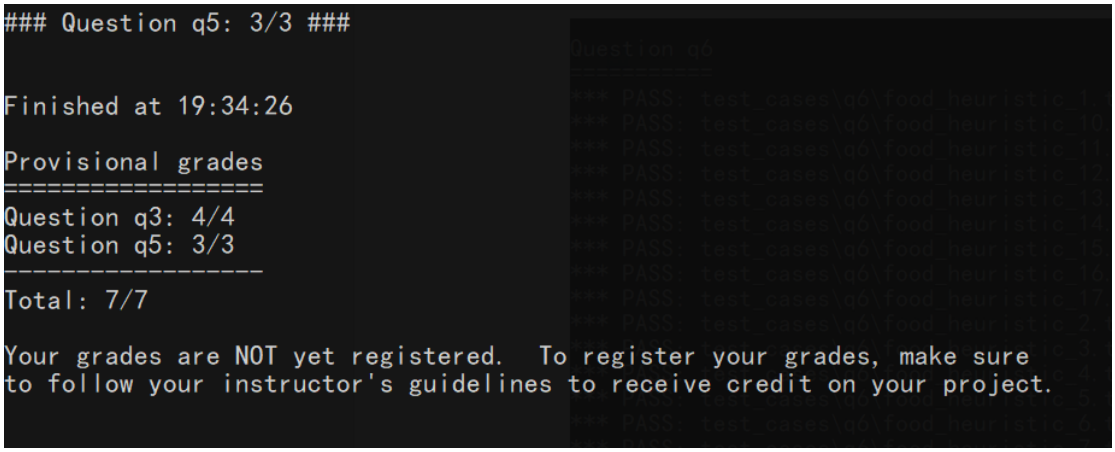

图 5：Question5 结果图

Question 6 (4 points): Eating All The Dots

实验记录：该问题为从起始点出发，吃掉 maze 中所有的点。该问题其实又是 Q5 的更难版本。但是我们认为仍然可以采取解决 Q5 类似的步骤解决该问题，但是，考虑到此时的 dots 一般会比较多（至少会比 Q5 中的 corners 多），所以，我们在解决 Q6 时不需要再使用 DFS 计算距离，而是直接使用城市距离即可。其他和 Q5 解决方案一样。

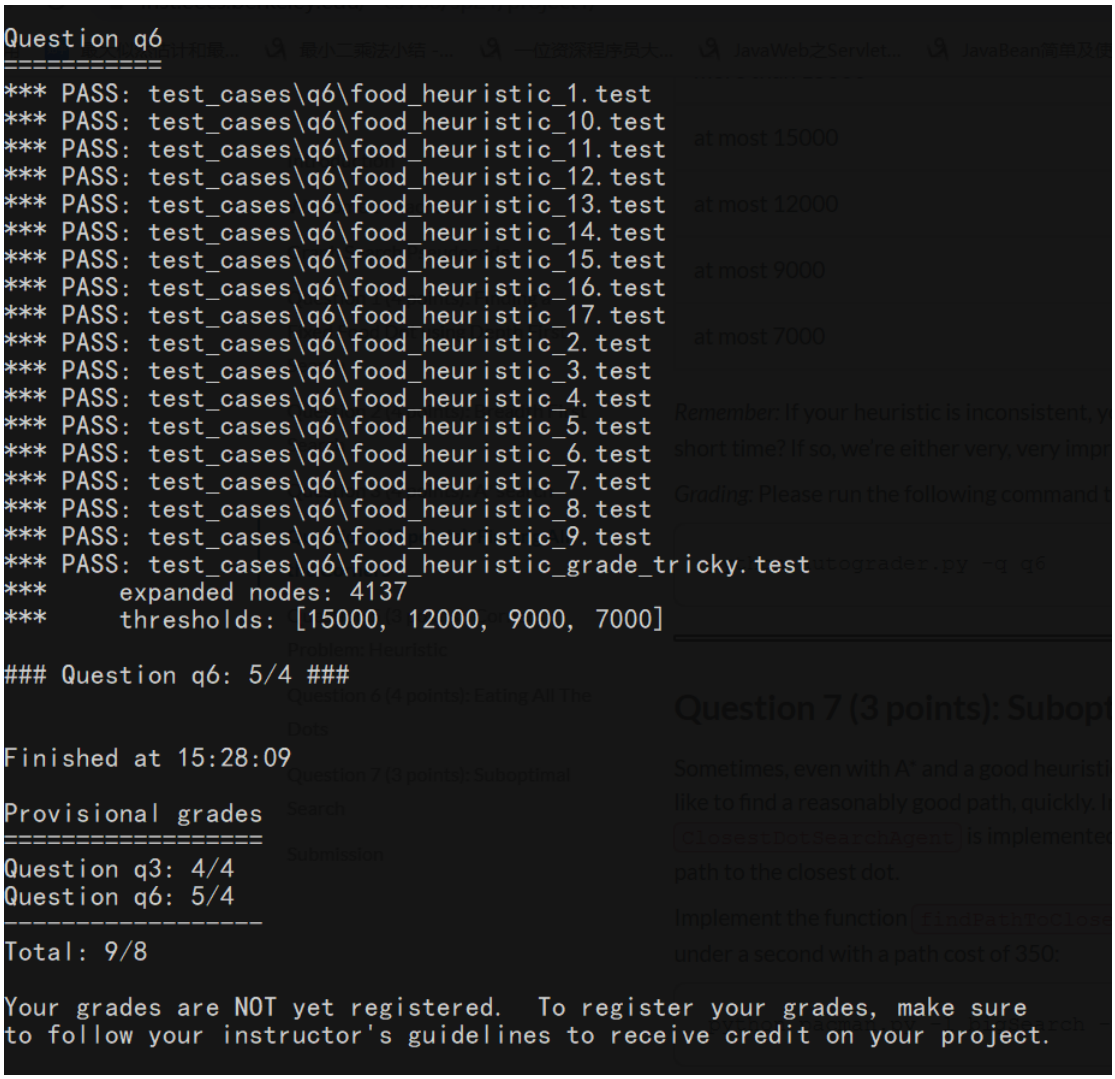

图 6：Question6 结果图

Question 7(3 points): Suboptimal Search

实验记录：在该问题中，根据问题描述，我们需要 always greedily 的吃 dot，因此，我们需要对 Q6 中的方案进行更改，在这里我们使用 A*算法来解决该问题，A*的启发信息和 Q3 中类似。

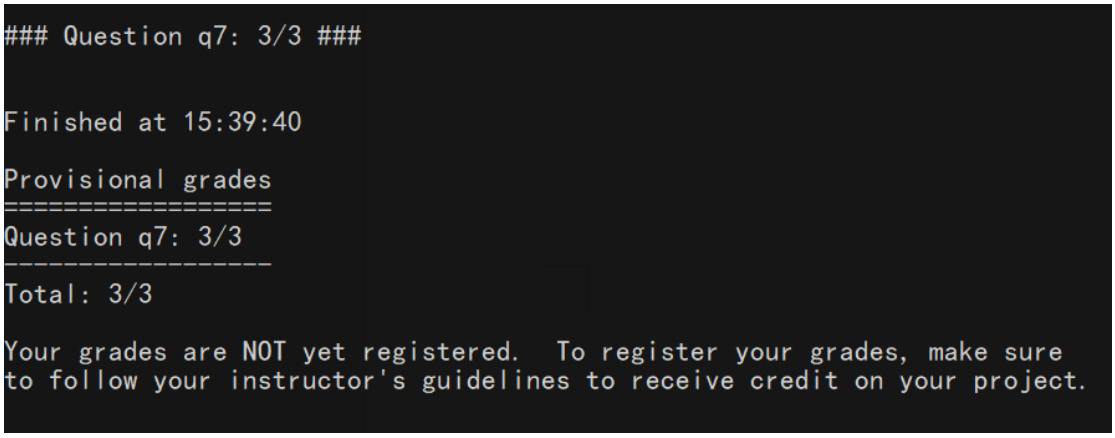

图 7：Question7 结果图