Department of Computer Science and Engineering

Islamic University of Technology (IUT)

A subsidiary organ of OIC

Laboratory Report

CSE 4618: Artificial Intelligence Lab

Name : Mohtasim Dipto
Student ID : 210042175
Section : BSc in SWE(A)
Semester : 6th
Lab No : 2

Introduction

Question1:

*We were asked to implement a function: A search algorithm, which took in a problem and the heuristic to determine the next successor picked to be explored.**
The heuristic function passed as a parameter takes the value of the null heuristic if no heuristic is provided in the parameter.

Question2:

This question asked for the definition of the problem and an implementation of the solution to the existing problem class `CornersProblem`.
In this problem, Pacman has to visit all four corners for the goal condition to be fulfilled.

Question3:

This question builds upon the previous task where it is asked to design a heuristic cornersHeuristic for the CornersProblem class that was admissible and consistent. The autograder provided us with the goals to meet for the task to be accepted.

Question4:

Similar to Question 3, we are asked to define an acceptable heuristic for a scenario where the Pacman has to eat all the food for the goal condition to be fulfilled. Once again, the autograder provided the goals the implementation has to meet to obtain full marks.

Question5:

This question asked for an implementation of a suboptimal solution and a suitable heuristic to an existing problem class AnyFoodSearchProblem, where the Pacman has to eat the food pellets but using a greedy algorithm, it eats the closest dots first. Two functions were implemented to solve this task.

Analysis & Explanation of the Problem

Question 1 Solution:

The implementation of the A* Search problem is similar to that of the search algorithms done in the previous lab. For a quick recap:

A fringe data structure is used to keep track of the nodes that needs to be visited which is updated every iteration with the successors of the currently visited node.

The root node is pushed into the fringe initially and a closed array is used to keep track of all the nodes that have been visited (thus it is initially empty as no nodes have been visited yet).

An iteration loops through all nodes stored in the fringe and follows a series of steps until the fringe is empty.

The current node (determined by the data structure of the fringe) is popped out of the fringe and it is checked whether it has already been visited or if it’s the goal state. For the former case, the current node is skipped and for the latter, the current node is returned as the solution.

If neither condition is true, successors of the current node are pushed into the fringe with any necessary information required for the problem to be solved.

This continues until we either obtain the solution or we have traversed through the entire fringe and not been able to obtain the solution. The generic steps of the algorithm are slightly modified for the A* Search, because the A* Search looks for the minimum cost path from initial state to goal state, thus cost is stored along with each node. This also requires use of priority queue where the elements are sorted by cost so that the node with minimum cost is picked first each time.

Although implementation is vastly similar to Uniform Cost Search algorithm, it differs because it considers both the forward cost required to reach the goal from that node and the cost of reaching that goal from the current node. The forward cost is calculated using a proper admissible heuristic function that is passed as a parameter.

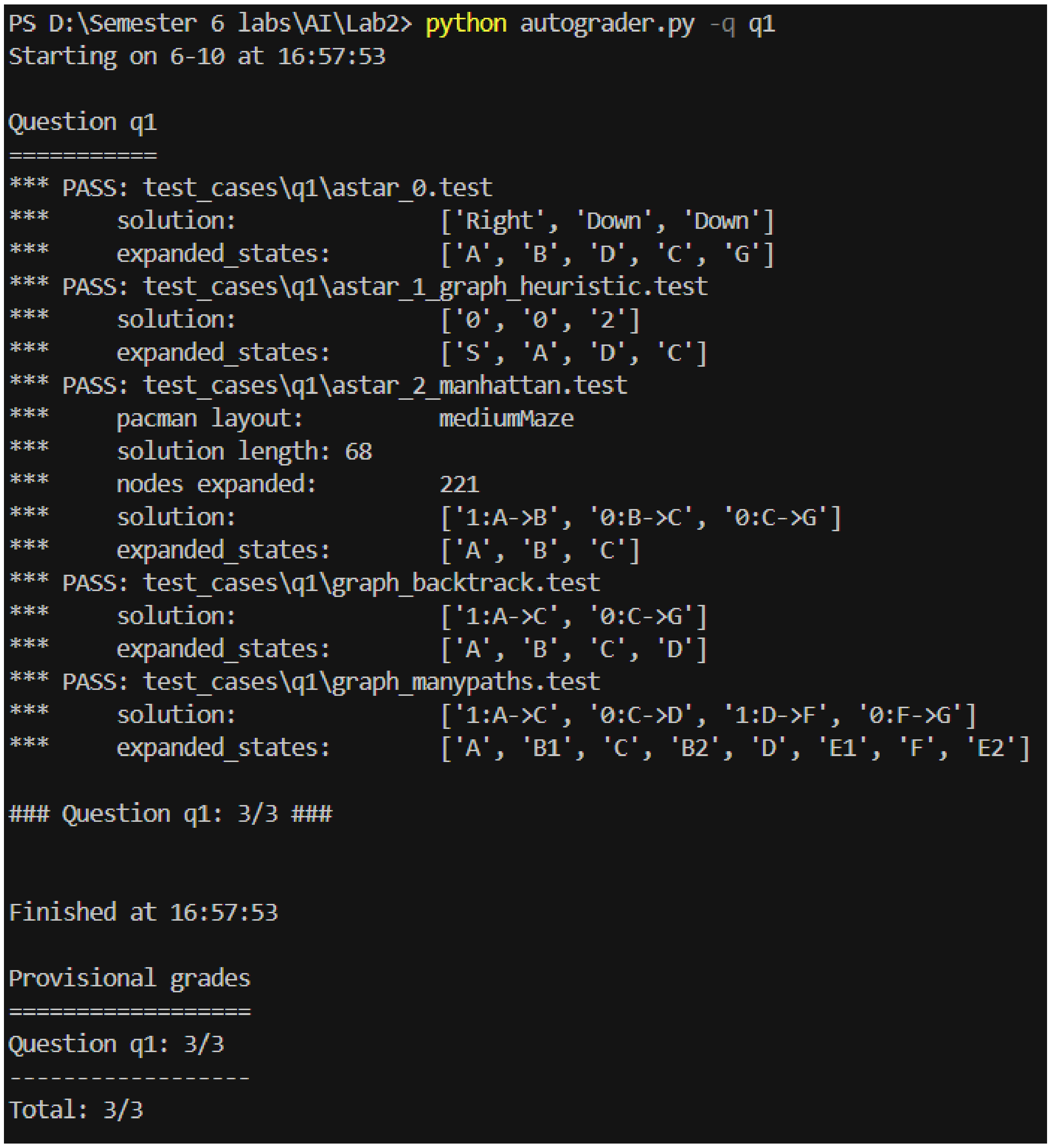
Explanation:

When this search function is called for a problem, the pacman's initial state is pushed into the priority queue. In the first iteration, all next possible states of the pacman from the root node are stored in the fringe by obtaining the successors of the root node. For each state, the heuristic function is used to get an approximate cost from the successor node to the goal node. The value obtained is added to the cost required to traverse from the root node to the successor node. Thus both the backward and forward cost is considered in this search condition. This is to ensure we only traverse the path which contains the least cost from the root node to goal node.

Calculation and consideration of the forward cost is what categorizes the A* Search algorithm as informed search. We consider the cost we might incur going forward using the heuristic function and the better our heuristic function is, the less nodes we have to expand because we reach and expand the least cost nodes first due to the priority queue structure.

For now, we assume the heuristic being passed as parameter is consistent and admissible and the heuristic values obtained are accurate enough to lead us to solution efficiently.

The root node is then added to the array keeping track of visited array. Thus in the future, if the root node is encountered again through some other path, it will not be visited again because we already know all the possible states from this node. This array, similar to in other search algorithms, reduces our number of nodes visited unnecessarily.



```
PS D:\Semester 6 labs\AI\Lab2> python autograder.py -q q1
Starting on 6-10 at 16:57:53

Question q1
===========
*** PASS: test_cases\q1\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q1\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q1\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q1\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']

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


Finished at 16:57:53

Provisional grades
==================
Question q1: 3/3
------------------
Total: 3/3
```

Question 2 Solution

For the problem, we were required to define the walls variable with the position of the game's environment as well as getting the Pacman's starting position. This is because the Pacman's movements cannot move through the walls so the walls have to be kept track of. The corners are then calculated through the position of the walls by considering the width of the walls as 1 and subtracting another 1 for the 0 indexing of the coordinates. Finally, for the solution of the game to be implemented a dictionary is initialized which stores a boolean value for each corner for keeping track of whether the corner has been visited or not. This is to be used later for calculating whether we have reached the goal state (and Pacman has successfully visited all four corners).

When the start state of the Pacman is obtained from the class, the starting position and the dictionary keeping track of visited corners is returned. Additionally, we also had to define an additional function for the test of goal checking to see whether the goal condition has been fulfilled. This function checks the dictionary to see if all the corners have been visited and if the condition is true, as per the game rules, the goal state is condition is fulfilled.

The class needed further definition of another function called getSuccessors which is required to get all the next possible nodes that can be visited from the current node. For this, we required knowledge of the predefined functions. The state whose successors needed to be obtained is received in parameter. Since we are storing the current position and the information about which corner has been visited, the state is unpacked to obtain this value. All possible actions, which are four actions in total, are explored to get the possible next state nodes. Using predefined functions, we obtain the change in coordinates when making a move. The change in coordinates is then added to get the coordinates of the next state of the Pacman, given the Pacman moves in that direction.

Since Pacman cannot move through walls, it is checked whether the next coordinates are that of the wall. If the Pacman does not move into wall for that corresponding action, the visited corners dictionary is copied for that state and based on whether the next node's coordinates matches that of a corner or not, the dictionary is updated and the successor is saved with that information.

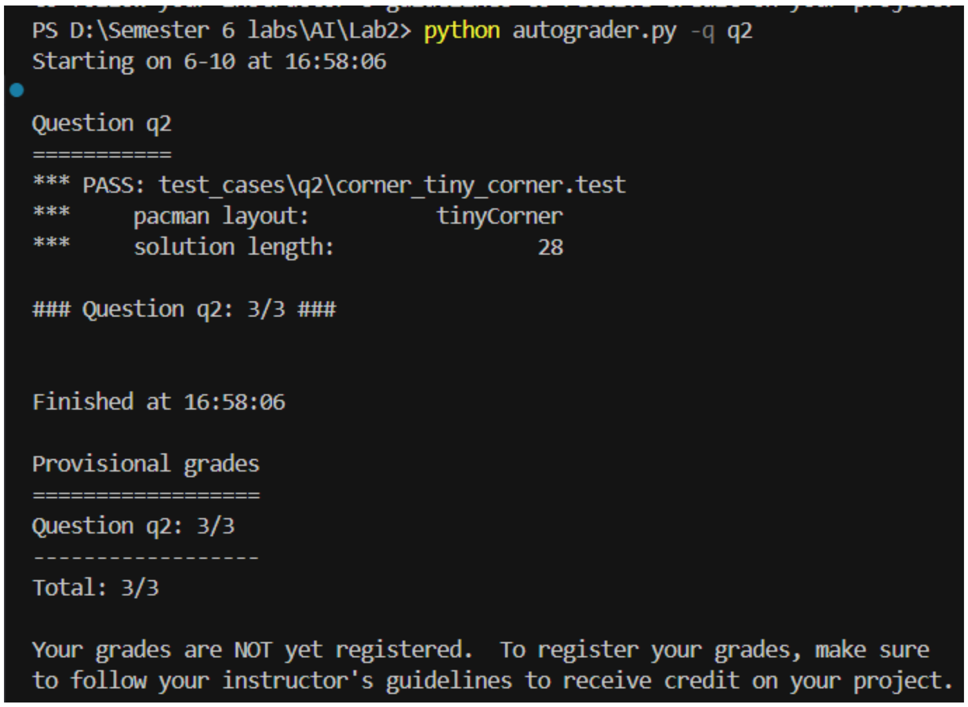
Explanation

The origin of the problem was that Pacman has to traverse through the game environment which consisted of walls and visit all four corners of the game. We were asked not to define the search algorithm this time, but to define the problem class's functions according to the requirements, which can later be used by any search algorithm.

For the initial states, the states/variables that had to be considered were the position of the walls since Pacman cannot move through walls and also the coordinates of the corners for checking whether Pacman has indeed visited all four corners. We also need to know where the Pacman's starting position is as well as keep track of which corners the Pacman has visited already so we can traverse optimally to the unvisited corners.

The problem class also needs a function to get the start state so we know where we started from at the beginning of the game. To know whether the game is over, we have to also define a function that can check whether the current state is indeed the goal state. For this, we can simply unpack the dictionary that is stored with node and if any particular node shows that Pacman has successfully visited all corners, then we can return that we have indeed reached the goal state and can stop the search algorithm.

As seen in any search algorithm, we have to know all possible moves from the current state to check whether any of the nodes will lead us to the goal state. This requires definition of yet another function which takes in the current node and explores all possible actions from that state. However, not all actions lead to valid states since the game environments also contains walls where the Pacman cannot end up in (since it cannot move through walls, as stated). At the end of the function, only the valid states are returned as successors. The states returned contain triplets to keep track of the next position and the corresponding dictionary for that position, action required to reach the successor and the cost of the action which is considered as 1 because all actions have the same cost.



```
PS D:\Semester 6 labs\AI\Lab2> python autograder.py -q q2
Starting on 6-10 at 16:58:06

Question q2
===========
*** PASS: test_cases\q2\corner_tiny_corner.test
***     pacman layout:          tinyCorner
***     solution length:                28

### Question q2: 3/3 ###


Finished at 16:58:06

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

Your grades are NOT yet registered.  To register your grades, make sure
to follow your instructor's guidelines to receive credit on your project.
```

Question 3 Solution

A function was defined to fulfill the objective of finding a proper heuristic for the previously defined problem scenario. The heuristic function takes in a problem and a particular node in that problem to find out the possible cost of reaching the goal from that node. Since we have stored the dictionary of visited corners with the state, we can simply unpack the state to obtain the dictionary to see which corners have been visited.

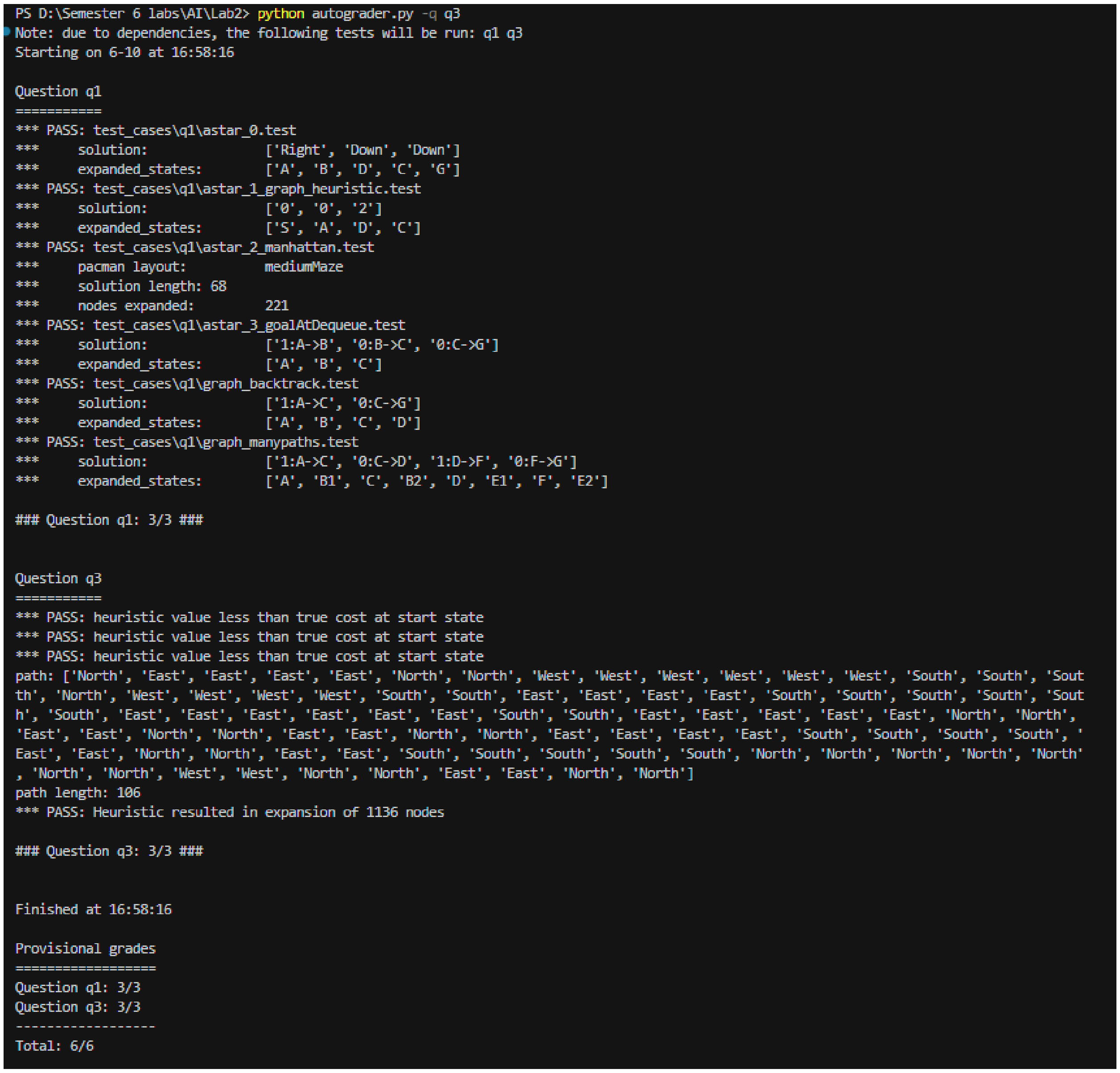
An array is initialized as empty and all the corners in the dictionary are iterated to find the coordinates of the corners for which the boolean is stored as false i.e. the corner has not been visited yet. The coordinates of the array is stored and manhattan distance is used to calculate how far the away the Pacman is in the current position to each corner that has not been visited yet. Next the maximum of the manhattan distances obtained is taken and returned as the heuristic value.

Explanation

The solution required us to find a heuristic that did not involve the calculation of finding the actual cost but provided a pretty good approximate cost. The heuristic value has to be smaller than actual cost but also consistent enough to reach the goal state optimally. In other words, we had to find the maximum cost Pacman would acquire to get to the next corner and thus take actions based on that. Our cost is based on the number of actions, hence we seek to estimate how many moves it might make to reach the next corner state which has not been visited yet.

Manhattan distance is an excellent approximate in this case since it just calculates the difference in y and x coordinates between two set of coordinates. It provides an underestimate because we are not considering that the Pacman might need to make more moves to avoid the walls. In fact this calculation does not take the position of the walls into consideration at all. Moreover, we take the maximum of these Manhattan distances as the heuristic cost because each Manhattan distance is an approximate of the cost to the next state and we want to take maximum of the approximate costs because the bigger the approximate cost (while still being underestimates), the closer they are to the actual costs. Thus the final value returned is an admissible and consistent heuristic.

There can be other approximate cost function similar to Manhattan that could possibly produce a good heuristic as well. For example, the maximum of the Euclidean distance from current state to next possible states or the maximum of the the values returned when calculating the maximum of the Euclidean and Manhattan distances. However, they produce more or less the same result and since Manhattan distance function was already defined, I used this function instead.



```
PS D:\Semester 6 labs\AI\Lab2> python autograder.py -q q3
Note: due to dependencies, the following tests will be run: q1 q3
Starting on 6-10 at 16:58:16

Question q1
===========
*** PASS: test_cases\q1\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q1\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q1\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q1\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q1\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']

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


Question q3
===========
*** PASS: heuristic value less than true cost at start state
*** PASS: heuristic value less than true cost at start state
*** 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', 'Sout
th', 'North', 'West', 'West', 'West', 'West', 'South', 'South', 'East', 'East', 'East', 'East', 'South', 'South', 'South', 'South', 'Sout
h', 'South', 'East', 'East', 'East', 'East', 'East', 'East', 'South', 'South', 'East', 'East', 'East', 'East', 'East', 'North', 'North',
'East', 'East', '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 1136 nodes

### Question q3: 3/3 ###


Finished at 16:58:16

Provisional grades
==================
Question q1: 3/3
Question q3: 3/3
------------------
Total: 6/6
```

Question 4 Solution

This problem class was already defined for a scenario where we had to define a function to find the heuristic cost of the distance from a given state to the closest food pellet. Since this problem is concerned with finding the food pellets, the state contains the current position and a grid containing the coordinates of food and whether each of them have been eaten or not.

The food grid is unpacked from the state provided as a parameter. Our objective is to find the approximate cost to the closest uneaten food pellet, thus an admissible and consistent heuristic. The coordinates of the uneaten food pellets is obtained using predefined function.

First it is checked whether we have already reached the goal state where no food pellets remains uneaten. If so, we return 0 as the Pacman does not have to move to position of the food pellet because there are no uneaten food left. Otherwise, we proceed with the next steps of the algorithm.

We first calculate the Manhattan distances from Pacman's current state to all the uneaten food coordinates. The minimum of these distances are taken which gives us the closest food pellet. Next an existing function mazeDistance is used to obtain the actual cost of the approximately closest food pellet from the current state. This gives us an approximate cost to the nearest food pellet.

Explanation

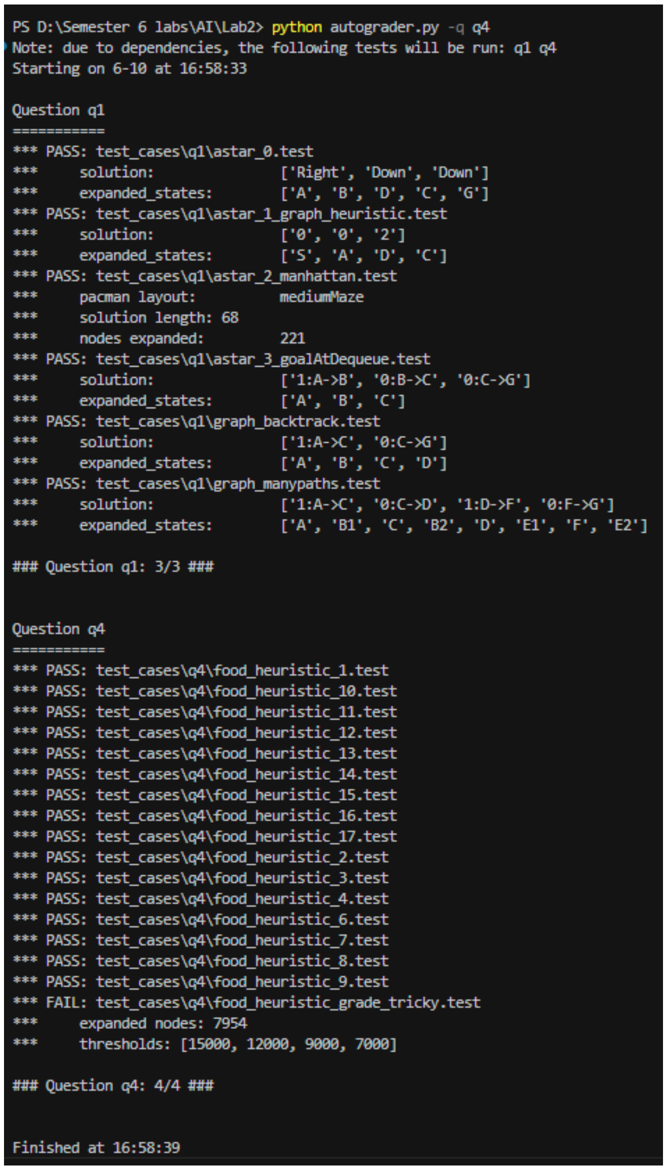
There were three possibilities to calculate the heuristic cost from current state to the uneaten food pellets:

Calculate the actual distances of all the uneaten food pellets from current coordinates and take the minimum of these to find which food pellet is actually closest. This would give us the best value since this is the actual value.

Calculate the approximate distance of all the uneaten food pellets from current coordinates and again, take the minimum to find the closest food pellet.

Before moving on to the third possible solution, the two other solutions can be explored a bit more. For the first solution, we calculate the actual distances which defeats the purpose of finding a heuristic. We find heuristic cost to avoid the complex and time consuming calculations of finding the actual cost. However, in the second solution, the distance received is an underestimate and in some cases, this ends up expanding more nodes which makes it less efficient and suboptimal for one of the cases of the autograder. This is because while it may find the closest food pellet, the distance it returns is too small and thus deviates from the actual cost by a lot. This means the heuristic is admissible certainly, but not consistent.

The third solution and the one I ended up following was the mixture of these two. We can avoid the complex calculations of finding the actual cost by graph traversal by simply finding the closest food pellet using the Manhattan distance approximation. Once we locate the closest food pellet's coordinates, we can use mazeDistance function to calculate the actual distance. Ultimately, we do end up calculating the actual cost but instead of doing this for all uneaten food pellets for every single state, we do this for one uneaten food pellet that is closest to our current state. This greatly reduces the time and resources while providing a good estimate of the cost. Here, we say "estimate" because the closest food pellet is found through approximate minimum Manhattan distance.



```
PS D:\Semester 6 labs\AI\Lab2> python autograder.py -q q4
Note: due to dependencies, the following tests will be run: q1 q4
Starting on 6-10 at 16:58:33

Question q1
===========
*** PASS: test_cases\q1\astar_0.test
***     solution:               ['Right', 'Down', 'Down']
***     expanded_states:        ['A', 'B', 'D', 'C', 'G']
*** PASS: test_cases\q1\astar_1_graph_heuristic.test
***     solution:               ['0', '0', '2']
***     expanded_states:        ['S', 'A', 'D', 'C']
*** PASS: test_cases\q1\astar_2_manhattan.test
***     pacman layout:          mediumMaze
***     solution length: 68
***     nodes expanded:         221
*** PASS: test_cases\q1\astar_3_goalAtDequeue.test
***     solution:               ['1:A->B', '0:B->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C']
*** PASS: test_cases\q1\graph_backtrack.test
***     solution:               ['1:A->C', '0:C->G']
***     expanded_states:        ['A', 'B', 'C', 'D']
*** PASS: test_cases\q1\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']

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


Question q4
===========
*** PASS: test_cases\q4\food_heuristic_1.test
*** PASS: test_cases\q4\food_heuristic_10.test
*** PASS: test_cases\q4\food_heuristic_11.test
*** PASS: test_cases\q4\food_heuristic_12.test
*** PASS: test_cases\q4\food_heuristic_13.test
*** PASS: test_cases\q4\food_heuristic_14.test
*** PASS: test_cases\q4\food_heuristic_15.test
*** PASS: test_cases\q4\food_heuristic_16.test
*** PASS: test_cases\q4\food_heuristic_17.test
*** PASS: test_cases\q4\food_heuristic_2.test
*** PASS: test_cases\q4\food_heuristic_3.test
*** PASS: test_cases\q4\food_heuristic_4.test
*** PASS: test_cases\q4\food_heuristic_6.test
*** PASS: test_cases\q4\food_heuristic_7.test
*** PASS: test_cases\q4\food_heuristic_8.test
*** PASS: test_cases\q4\food_heuristic_9.test
*** FAIL: test_cases\q4\food_heuristic_grade_tricky.test
***     expanded nodes: 7954
***     thresholds: [15000, 12000, 9000, 7000]

### Question q4: 4/4 ###


Finished at 16:58:39
```

Question 5:

Solution

This involves defining a function to check whether, for the scenario described in previous two problems, the Pacman has reached the goal state. Using this function, we were asked to define an agent that eats the closest dots first.

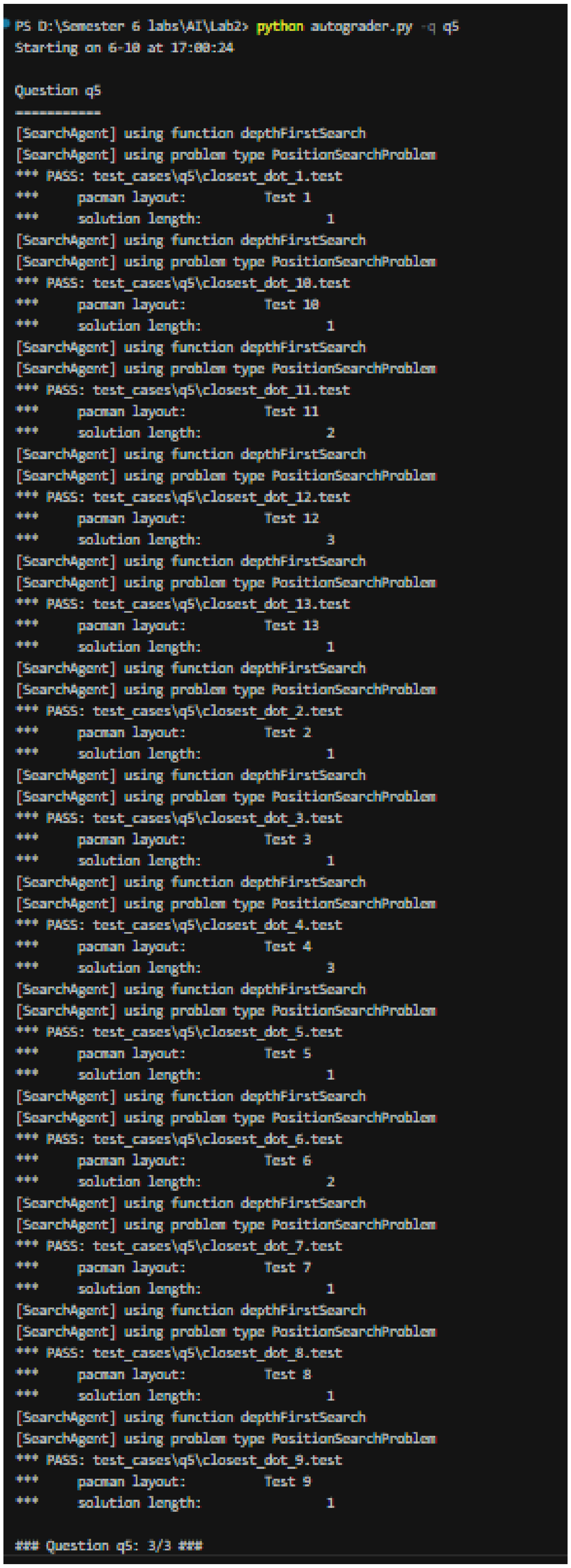
The first function took in the current state to check whether Pacman has reached the goal, which is to have eaten a food pellet. The state is unpacked to find the current x and y coordinates and the food grid is checked to see whether Pacman is currently situated in a position where there is a food pellet. The x and y coordinates are checked in food grid to see if there had been an uneaten food pellet. If so, then Pacman has just eaten a food pellet which fulfills our goal condition and so we can return true.

The second function required using the goal check function to see whether Pacman's search for the closest food pellet has led to the goal state. The search algorithm was implemented by calling the UCS implemented in the previous lab task. When the goal check function returns true, the function returns the sequence of actions that led up to the goal state.

Explanation

The task was to find a suboptimal greedy search to find the closest dot at first. This is because the original problem was to find the optimal path through all the food pellets. For large problems, this may end up taking too much time and resources, thus we were asked to design the search such that the closest food pellet is found each time and then from then on, we check if we reached the goal state i.e. if all the food has been eaten already. In this way, we don't have to compute the entire optimal path at once.

The reason for including UCS was that we were not required to find optimal solution, thus we could forego the heuristic calculations and simply consider the cost it takes to move from one state to another. UCS uses a priority queue so the nodes with the least cost/distance from the current node is explored first until we find a goal state that can be reached using the nodes with minimum cost edges, which fulfilled the objective asked.



```
PS D:\Semester 6 labs\AI\Lab2> python autograder.py -q q5
Starting on 6-10 at 17:00:24

Question q5
===========
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_1.test
***     pacman layout:          Test 1
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_10.test
***     pacman layout:          Test 10
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_11.test
***     pacman layout:          Test 11
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_12.test
***     pacman layout:          Test 12
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_13.test
***     pacman layout:          Test 13
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_2.test
***     pacman layout:          Test 2
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_3.test
***     pacman layout:          Test 3
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_4.test
***     pacman layout:          Test 4
***     solution length:                3
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_5.test
***     pacman layout:          Test 5
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_6.test
***     pacman layout:          Test 6
***     solution length:                2
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_7.test
***     pacman layout:          Test 7
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_8.test
***     pacman layout:          Test 8
***     solution length:                1
[SearchAgent] using function depthFirstSearch
[SearchAgent] using problem type PositionSearchProblem
*** PASS: test_cases\q5\closest_dot_9.test
***     pacman layout:          Test 9
***     solution length:                1

### Question q5: 3/3 ###
```

Interesting Findings and Challenges

Question 1
One of the interesting observations was if we used a trivial heuristic which returned 0, it turns into a UCS where the nodes are explored in the order of the total cost from the start node. This is because the heuristic cost, which is 0, becomes useless, always returning zero.
This is the only advantage A* has over UCS and the factor that makes it "informed" and thus optimal search — because it can somewhat look into the future, something UCS is unable to do.

Question 2
This problem scenario was easy to understand but the implementation was difficult to implement. Alsounderstanding the underlying predefined functions required to complete the implementation made it confusing at times.

Question 3
The approximation of the costs between states was done using Manhattan cost, as it is a standard method for calculation of heuristics. However, when it came to what to do with those values, there was a bit of confusion on my part.
This is because admissible heuristics are supposed to be an underestimate. This led me to believe the minimum of the distances had to be returned as heuristics, which gave me partial marks in the autograder.
There were a couple of other approaches attempted before the concept was clear: the Manhattan distances were all underestimates. If the **maximum** of these values are taken, the heuristic would still give the underestimate. This approach rewarded me with full marks from the autograder.

Question 4
This heuristic was a bit trickier than the previous one. As stated in the explanation, I tried two methods before realizing what was needed was a mixture of the two.
Using **actual costs** as heuristic is the best heuristic, as it uses the actual cost to make informed decisions which are better.
Using the Manhattan distance, Euclidean distance, or any other methods to approximate the distance led to partial marks because they led to unnecessary

expansion of nodes. They did not come close enough to the actual heuristic.
This solution required a bit of research and trial and error on my part since I wanted to find how to calculate a good heuristic without calculating the actual cost for all the uneaten food from the current state.

Question 5

The problem itself asked to find a greedy solution, and it was a bit challenging what to use to implement the greedy search. However, by process of elimination, DFS and BFS were ruled out since they do not consider the cost/distance from current states to food pellets.
The other choices were UCS and A* Search algorithms. UCS is simple to implement since it does not require any heuristic, which the class does not provide, and thus it was the obvious winner.
An interesting finding for this solution was this same result could be obtained using A* Search algorithm with a poor heuristic, which could lead to suboptimal paths and lead to a suboptimal solution to the closest food pellet, as was the objective of the function.