Ονοματεπώνυμο: Παντελεήμων Μαλέκας
A.M: 1115201600268

Comments regarding assignment 1 (Pacman)

- Q1: Depth First Search

The general idea for all algorithms works as follows: The starting state is inserted into the fringe. Then, we enter a loop where we remove a node from the fringe, we get its successor states, and we insert them into the fringe. If we pop a state that is already visited, we skip it and pop the next node from the fringe. If we pop the goal state, we end the loop and start building the path that will be returned. If a successor state is already visited it won't be inserted into the fringe. In addition, to keep some helpful information regarding each state, I used a Node class which contains the parent state, the action that brought us to the state e.t.c.

Since the fringe in DFS follows a LIFO policy I used the Stack structure from util.py

To build the path, after the loop has finished, we get the action of the last state that was popped from the fringe and insert it into a path list. Then we get the parent of each state all the way back to the root adding each action at the start of the path. If we reach a "None" action, the loop finishes, so we return the path. This is common for all algorithms.

- Q2: Breadth First Search

For BFS, the idea is pretty much the exact same thing as in DFS except I used a Queue for the fringe, since the algorithm follows a FIFO policy.

- Q3: Uniform Cost Search

Similar idea also implemented, except I used the PriorityQueue structure in order to prioritize the actions with the smaller costs.

- Q4: A* Seach

PriorityQueue is also used here, only it prioritizes the sum of both the cost and the result of the heuristic function, since we're using A*. Each time a state is about to be inserted into the fringe, we call the heuristic function and add its result along with the cost.

- Q5: Corners Problem: Representation

The idea for the Corners Problem: Each state is comprised of the coordinates of Pacman's position and the corners that Pacman has not travelled to. For the starting state we place all the corners as part of the state since Pacman hasn't reached any corner yet.

When we check if we reached the goal state, we check if the current position is one of the goals and we don't have any other corners remaining. If so, this means we have reached our goal state so the algorithm ends here.
Regarding the successor states: We copy the list of our current remaining corners, to see what will be passed in the successor state. The list will be updated if we've reached a corner, or it will remain the same if we haven't. We check if the successor's position is one of the corners and if it's one of the corners where the current state hasn't reached yet. If so, this means we have reached one of the corners so we remove it from the list. This way, the successor state will know that Pacman has reached one of the desired corners. So the successor state is comprised of the new coordinates along with the remaining corners.

- Q6: Corners Problem: Heuristic

I used the manhattanDistance function from util.py so we can calculate the desired distances. Since we have a grind and 4 available directions, the Manhattan Distance is our best choice.
If we have no remaining corners we have reached our goal state, so the heuristic function returns 0. Otherwise, we get the shortest Manhattan Distance from our current position to the remaining corners. In addition, we have to calculate the shortest Manhattan Distances from each corner combination and add them. For example: if we have corners [a,b,c,d], we get the shortest of distances (a,b), (a,c), (a,d), the shortest of (b,c), (b,d), and the only distance of (c,d). Since our problem is similar to the shortest path problem, I used a logic similar to Dijkstra's algorithm to get the desired distances. So the heuristic function returns the sum of all these distances along with the first distance we got from the current position.

- Q7: Eating All The Dots: Heuristic

Here, we get a list of our remaining food. If the length of this list is equal to 0, this means we have no remaining dots, so we have reached our goal state, so the heuristic function returns 0. For this heuristic, I used the mazeDistance function provided in this file. Manhattan Distances didn't provide good enough solutions and since mazeDistance can be used here, it was a better option overall. In addition, mazeDistance takes care of the walls in the maze, contrary to the manhattanDistance function. So, first we get the distance from our current position to the furthest food. I tried minimum distance at first, but since we're dealing with a problem similar to the Travelling Salesman Problem, the longest distance is preferable, contrary to the shortest one as we used in cornersHeuristic. In addition, we also get the longest distance from each food combination. Similar to what I did in cornersHeuristic. If the longest distance of the current combinations is larger than the one from currentState to the furthest food, or larger than the distance from the previous combinations, it is a preferable solution so we return it as the result of the heuristic function.

- Q8: Suboptimal Search

In the goal test we get the coordinates of the state, as well as the food grid. We check if we still have food at our current state. If not, we have reached the goal state. For finding the path to closest dot I used the UCS algorithm since it greedily moves to the first available state.