CS 188 Fall 2024 Pacman Project

Project 1

Question 1-4

I designed the search algorithms based on their theoretica counter parts.

1. `depthFirstSearch` may have many implementations, but I prefer the one using a stack for a search frontier.
2. `breadthFirstSearch` is implemented with a queue as a search frontier.
3. `uniformCostSearch` uses costs so in fact I am implementing a shortest path algoirthm. Costs are always positive, so I implemented Dijkstra's algorithm, using a priority queue as a search frontier.
4. `aStarSearch` with a `nullHeuristic` is a `uniformCostSearch`. I based my implementation on `uniformCostSearch`, but with taking account the heuristic when updating the costs of the paths explored.

I had to figure out that successors do not only return states, but actions and costs. I take that into account when forming my stack, queue or priority queue in each algorithm. In particular, for questions 1 and 2, I am interested in the actions returned by successors, as this is what the algorithm shall return. In questions 3 and 4 I need the cost as well.

In theory these algorithms can traverse the whole graph (problem), so I manually include a termination condition, based on the `isGoalState` method of a problem.

Question 5

I define the state of a `CornersProblem` as the pacman position together with a quadraple of booleans representing which of the corners are visited. I would prefer to use a dictionary, but the `uniformCostSearch` algorigthm may fail, because it uses a dictionary of costs with the states as keys, so I want the states to be hashable. The initial state then is `self.StartingPosition, (False, False, False, False)`.

Note however that in `CornersProblem.getSuccessors`, I still prefer the `dict` solution, so I align the tuple boolean values with `self.corners`, so that I can easily update which corner is visited. Then I simply get a tuple copy of the `dict` values for the new state.

Question 6

Assume a current position and a few corners left to visit (maybe all or maybe none). I find the closest by Manhattan distance to any of them. Then I wand to find the next closest by Manhattan disance from that point. And repeat until no corners are left. Basically, I create a tesselate path of straight (by Manhattan logic) lines from the current position and through all (remaining) corners.

Question 7

At first I tried the simple `foodGrid.count()` because pacman will need at least this many steps to eat all remaining food, assuming all food is connected and in a path. This scored a mere `2/4` with the autograder.

I then decided to expand the first step by adding the entire Manhattan distance of pacman to the nearest food. Calculating distances for all foods seems inexpensive, as the autograder finishes quickly with a mere `3/4`.

I then decided to use `mazeDistance` instead of Manhattan distance, found in `searchAgents.py`. However this is very slow, probably because `mazeDistance` uses a search algorithm on each call (I think). It also scored a `3/4`.

Then I decided to choose the nearest food by Manhattan distance, but count the maze path to it at the end, so one maze path cannot be that expensive. This heuristic is still admissible, because the worst case scenario that all food is in a path and the closest one is the first one, the maze distance to it cannot include any food on the way, so the total steps to eat all the food are not overestimated. This solution scored a mere `5/4`!

Question 8

The problem is already well defined in `AnyFoodSearchProblem`. All we need is find a food as a goal, and let a search algorithm find the optimal path to it. Because the goal state is *any* food to be found, this means that the optimal path to any food is the closest food path. So it is not searching a goal to a specific position, I hit the goal as soon as wherever I am as a pacman, there is food.

By well-defining the problem, the solution then for finding the closest food in `ClosestDotSearchAgent.findPathToClosestDot` is simply to use a serach algorithm from the ones implemented, as they already return actions. I chose the `uniformCostSearch`.