Artificial Intelligence course, **2019-2020**

Exercise 1

The Pacman Project 1

by “Berkeley University”

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

The **Depth First Search algorithm (DFS)** searches the Pacman’s board in depth in order to construct the path to the goal state.

The implementation of the DFS goes as following:

1) A **Stack** data structure, called “frontier”, is utilized to hold the unvisited nodes expanded by the algorithm.
2) The starting node is pushed in the frontier.
3) A **Set** data structure implemented internally as a **Hash-Table** to provide O(1) access. This is utilized to hold the “visited locations” of the board.
4) While the frontier is not empty:
    a. Pop a node from the frontier-stack.
    b. Check if the location of the popped node is the **Goal State**, if it is then exit immediately by return the path to this location.
    c. Add this location to the visited ones.
    d. For each successor of this location. if the successor’s location is not in the visited ones, then push the successor’s location and the cumulative path to the frontier.
5) If the frontier is found empty before the goal state has been reached, then return “None” to indicate failure.

We evaluate that the DFS is well implemented by running the following tests:

1) `python pacman.py -l tinyMaze -p SearchAgent`

2) `python pacman.py -l mediumMaze -p SearchAgent`

   Here we get a solution which has a length of 130, which is the best-case scenario for DFS when using the **Stack** data structure and pushing successors onto the frontier in the order provided by "getSuccessors()" method.

   This is not the general-best-case scenario as it's clear from the graphical red board of Pacman. The optimal solution for this test-case is provided by the BFS (see question 2).

3) `python pacman.py -l bigMaze -z .5 -p SearchAgent --frameTime 0`

Question 2: Breadth First Search (BFS)

The **Breadth First Search algorithm (BFS)** searches the Pacman's board in breadth in order to construct the path to the goal state.

The implementation of the BFS goes as following:

1) A **Queue** data structure, called "frontier", is utilized to hold the unvisited nodes expanded by the algorithm.
2) The starting node is pushed in the frontier.
3) A **Set** data structure implemented internally as a **Hash-Table** to provide O(1) access. This is utilized to hold the "visited locations" of the board.

4) While the frontier is not empty:
    a. Pop a node from the frontier-queue.
    b. Check if the location of the popped node is the **Goal State**, if it is then exit immediately by return the path to this location.
    c. If this location is not in the visited ones, then:
        i. Add the location to the set.
        ii. Foreach successor of this location, push the successor's location and the cumulative path to the frontier.
5) If the frontier is found empty before the goal state has been reached, then return "None" to indicate failure.

The **BFS differs from the DFS** in two key points:

1) The frontier in BFS uses a queue (FIFO) data structure instead of a stack (LIFO).
2) The popped node's successors in BFS, are all pushed to the frontier, if the current location is not already visited. Instead, in the DFS, each successor-node has to not be already visited in order to be pushed.

We evaluate that the BFS is well implemented by running the following tests:

1) `python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs`
2) `python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5 --frameTime 0`

The implemented BFS code is a general-case BFS, so apart from Pacman, it also solves the "eight-puzzle search problem". This is easily observed by running:

`python eightpuzzle.py`

Question 3: Varying the Cost Function

The **Uniform Cost Search algorithm (UCS)** searches the Goal State in Pacman's board by using prioritization based on cost.

The implementation of the UCS goes as following:

1) A **Priority Queue** data structure, called "frontier", is utilized to hold the unvisited nodes expanded by the algorithm.
2) The starting node is pushed in the frontier.
3) A **Set** data structure implemented internally as a **Hash-Table** to provide O(1) access. This is utilized to hold the "visited locations" of the board.
4) While the frontier is not empty:
    a. Pop a node from the frontier-priority-queue.
    b. Check if the location of the popped node is the **Goal State**, if it is then exit immediately by return the path to this location.
    c. If this location is not in the visited ones, then:
        i. Add the location to the set.
        ii. Foreach successor of this location, push a) the node consisting of the successor's location, the cumulative path and the cumulative cost to the frontier, b) the cumulative cost separately.
5) If the frontier is found empty before the goal state has been reached, then return "None" to indicate failure.

We evaluate that the UCS is well implemented by running the following tests:

1) `python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs`
2) `python pacman.py -l mediumDottedMaze -p StayEastSearchAgent --frameTime 0`
3) `python pacman.py -l mediumScaryMaze -p StayWestSearchAgent --frameTime 0`

Question 4: A* search

The **A* Search algorithm** searches the Goal State in Pacman's board by using a heuristic function.

The implementation of the **A* Search** goes as following:

1) A **Priority Queue** data structure, called "frontier", is utilized to hold the unvisited nodes expanded by the algorithm.
2) The starting node is pushed in the frontier.
3) A **Set** data structure implemented internally as a **Hash-Table** to provide O(1) access. This is utilized to hold the "visited locations" of the board.
4) While the frontier is not empty:
    a. Pop a node from the frontier-priority-queue.
    b. Check if the location of the popped node is the **Goal State**, if it is then exit immediately by return the path to this location.
    c. If this location is not in the visited ones, then:
        i. Add the location to the set.
        ii. Foreach successor of this location, push a) the node consisting of the successor's location, the cumulative path and the cumulative cost to the frontier, b) the sum of the cumulative cost of the node with the heuristic cost calculated.
5) If the frontier is found empty before the goal state has been reached, then return "None" to indicate failure.

The **A* Search differs from the UCS** in one key point: when pushing to the frontier, the general cost to that node is calculated as the sum of the cumulative cost to that node with the calculated heuristic cost.

Instead, in UCS, only the cumulative cost of the node is assumed as the general cost.

We evaluate that the A* is well implemented by running the following tests:

1) `python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic --frameTime 0`

2) `python pacman.py -l openMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic`

Question 5: Finding All the Corners

In this problem, the **Goal State** is the state in which the Pacman has eat all of the dots found on the corners.

In order for the corners-problem to be solved, there is the need for the following class-methods to be implemented:

1) getStartState(): return the startingPosition along with the corners.
2) isGoalState(): return “True” if the length of the corners-set is zero. We know that upon reaching the goal state, all the corners will have been removed.
3) getSuccessors():
    a. for each one of the four directions (NORTH, SOUTH, EAST, WEST), calculate the destination-coordinates and if they are not leading to a wall then:
        i. Save the “remainingCorners” and the “newPosition”.
        ii. The “remainingCorners” is a “tuple”-data structure and no element can be removed directly. So, we want to check if the “newPosition” is included in the “remainingCorners” and then:
            1. Save the “remainingCorners”, which is a “tuple” to a temporary list.
            2. Remove the “newPosition” from that temporary list.
            3. Set the “remainingCorners” to be a tuple of the temporary list.
    b. Increment the “expandedNodes”-counter and return the list of successors.

For this solution to work, the **Breadth First Search** Algorithm was implemented first (question 2).

We evaluate that the A* is well implemented by running the following tests:

1) `python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`

   The shortest path through tinyCorners takes 28 steps as requested by the specs.

2) `python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem`

3) `python autograder.py -p --test test_cases/q5/corner_tiny_corner`

Question 6: Corners Problem: Heuristic

In order for the corners-problem to be solved with a heuristic approach, there is the need for the “cornersHeuristic”-method to be implemented as follows:

In short:

Iterate over the "remainingCorners", find the "bestNextCorner" add its cost to the “sum” and remove the corner from the “remainingCorners”. Loop until no remaining corners exist. Return the “sum”.

In details:

1) Initialize the “sum” to zero and save the “currentPosition” and the “remainingCorners”-tuple.

2) While there are “remainingCorners”:
    a. Assume the first remaining corner to be the lower-cost-corner thus the “bestNextCorner”.
    b. Iterate over the “remainingCorners” (starting from the 2nd one) and find the actual minimum-cost corner.
    c. Add that min-cost value to the “sum”, which will be returned in the end.
    d. Delete the “bestNextCorner” from the “remainingCorners” by using the help of a temporary set-data-structure (as no elements can be removed directly from a tuple).
    e. Update the “currentPosition”.
3) Return the “sum”.

We evaluate that the “cornersHeuristic” is well implemented by running the following tests:

1) `python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5`

   The expanded nodes in “mediumCorners” are 692, nearly half of the best-case limit of 1200 nodes.

2) `python pacman.py -l bigCorners -p AStarCornersAgent -z 0.5 --frameTime 0`

For this solution to work, the **A* search** Algorithm was implemented first (question 4).

Question 7: Eating All The Dots

The Goal State in this problem is for Pacman to eat all the “food”-dots by using a heuristic function.

The “foodHeuristic” function is implemented as follows:

1) Converts the “foodGrid” to a list.
2) If the list does not contain any elements, then we have reached the Goal State, in which no food-dots exist.

3) Assume the first food-dot to have the maximum travel-cost from the Pacman's position.
4) Iterate over the food-dosts (starting from the 2nd one) and find the actual maximum-cost food-dot.
5) Return the maximum-cost.

We evaluate that the "foodHeuristic" is well implemented by running the following tests:

1) python pacman.py -l tinySearch -p AStarFoodSearchAgent

2) python pacman.py -l trickySearch -p AStarFoodSearchAgent

   The total cost for the path found by this test is 60. It was found in 3.9 seconds.

   The expanded nodes were 9551.

For this solution to work, the **A* search** Algorithm was implemented first (question 4).

Question 8: Suboptimal Search

The Goal State in this problem is for Pacman to reach the closest dot each time. This is a Greedy approach to the "Eating all dots" problem. The home is that by eating always the closest dot, in the end it will have eat all the dots in the minimum time and by expanding the minimum number of nodes.

The solution is implemented as follows:

1) The “isGoalState”-method of “AnyFoodSearchProblem”-class gets set to return “True” if Pacman sits on a food-dot or “False” if it does not.
2) The “findPathToClosestDot”-method of “ClosestDotSearchAgent”-class is set to return the path found by the BFS algorithm.

We evaluate that the “findPathToClosestDot” is well implemented by running the following test:

```
python pacman.py -l bigSearch -p ClosestDotSearchAgent -z .5 --
frameTime 0
```

It is found that the above test-case works equally well the same when either BFS, UCS or A* are used in “findPathToClosestDot”-method, but it does not work at all if DFS is used, that’s because the DFS is returning a huge path.

The “ClosestDotSearchAgent” won't always find the shortest possible path through the maze because even though it repeatedly goes to the closest dot.

One example explaining this case is when by following the closest dots Pacman leaves behind a handful of dots which later become a very distant ones.

If Pacman wasn’t going for the closest dot but its goal was to “clear an area of close dots” it wouldn’t have to travel far in the end just to collect a couple of dots left behind.

This problem is visible when running the above test-case (“bigSearch”).