Artificial Intelligence
Report for Project 01: The Searching Pac-Man

Question 1 (10 points) Implement the depth-first search (DFS) algorithm in the depthFirstSearch function in search.py. To make your algorithm *complete*, write the graph search version of DFS, which avoids expanding any already visited states (R&N 3ed Section 3.3, Figure 3.7).

Analysis:

- DFS was implemented using the stack data structure.
- Each stack node consists of a tuple of (***state, actionList, cost***), where ***state*** represents the Pac-Man position, ***actionList*** represents a list of actions required to get to that ***state*** and ***cost*** represents the cost to reach that state.

Depth First Search				
Command	**Nodes Expanded**	**Memory Usage**	**Running Time (in sec)**	**Critical Analysis**
python pacman.py -l tinyMaze -p SearchAgent	15	O(bm) where b= brancing factor of the DFS tree and m is the depth of the DFS tree.	0	PacMan found the path with a cost of 10 and a score of 500.
python pacman.py -l mediumMaze -p SearchAgent	146		0	PacMan found the path with a cost of 130 and a score of 380.
python pacman.py -l bigMaze -z .5 -p SearchAgent	390		0	PacMan found the path with a cost of 210 and a score of 300.

Question 2 (10 points) Implement the breadth-first search (BFS) algorithm in the breadthFirstSearch function in search.py. Again, write a graph search algorithm that avoids expanding any already visited states. Test your code the same way you did for depth-first search.

Analysis:

- BFS was implemented using the queue data structure.

- Each queue node consists of a tuple of (***state, actionList, cost***), where ***state*** represents the Pac-Man position, ***actionList*** represents a list of actions required to get to that ***state*** and ***cost*** represents the cost to reach that state.

Breadth First Search				
Command	**Nodes Expanded**	**Memory Usage**	**Running Time (in sec)**	**Critical Analysis**
python pacman.py -l mediumMaze -p SearchAgent -a fn=bfs	269	$O(b^m)$ where b= brancing factor of the BFS tree and m is the depth of the BFS tree.	0	PacMan found the path with a cost of 68 and a score of 442.
python pacman.py -l bigMaze -p SearchAgent -a fn=bfs -z .5	620		0	PacMan found the path with a cost of 210 and a score of 300.

Question 3 (10 points) Implement the uniform-cost search (UCS) algorithm in the uniformCostSearch function in search.py.

Analysis:

- UCS was implemented using the priority queue data structure to obtain the next node with the least cost.
- Each priority node consists of a tuple of ((***state, actionList, cost***), ***priority***) where ***state*** represents the Pac-Man position, ***actionList*** represents a list of actions required to get to that ***state***, ***cost*** represents the cost to reach that state and ***priority*** represents the priority of that node which for this case is equal to the ***cost*** value.

Uniform Cost Search				
Command	**Nodes Expanded**	**Memory Usage**	**Running Time (in sec)**	**Critical Analysis**
python pacman.py -l mediumMaze -p SearchAgent -a fn=ucs	269	$O(b^{C^*/e})$ where b= brancing factor of the UCS tree, C* is the optimal solution cost and e is the minimum edge cost in the UCS tree.	0	PacMan found the path with a cost of 68 and a score of 442.
python pacman.py -l mediumMaze -p SearchAgent	146		0	PacMan found the path with a cost of 130 and a score of 380.
python pacman.py -l bigMaze -z .5 -p SearchAgent	390		0	PacMan found the path with a cost of 210 and a score of 300.

Question 4 (15 points) Implement A* graph search in the empty function aStarSearch in search.py. A* takes a heuristic function as an argument.

Analysis:

- aStarSearch was implemented using the priority queue data structure that makes use of a heuristic function along with the cost as the priority value to obtain the next node with least cost.
- Each priority node consists of a tuple of ((***state, actionList, cost***), ***priority***) where ***state*** represents the Pac-Man position, ***actionList*** represents a list of actions required to get to that ***state***, ***cost*** represents the cost to reach that state and ***priority*** is the sum of ***heuristic function value + cost value.***

aStar Search				
Command	**Nodes Expanded**	**Memory Usage**	**Running Time (in sec)**	**Critical Analysis**
python pacman.py -l bigMaze -z .5 -p SearchAgent -a fn=astar,heuristic=manhattanHeuristic	549	O(n) where n is the number of different positions in the grid.	0	PacMan found the path with a cost of 210 and a score of 300.

Question 5 (10 points) Implement the CornersProblem search problem in searchAgents.py.

- The solution uses **queue data structure** to solve the problem.
- The state that is stored in the queue is **[(corner1), (corner2), (corner3), (corner4), (Current state), ActionList, Cost]**
- It helps to check that through this path which corners are visited. The state that have all the corners as visited, is the fastest path and the corresponding action list is returned.

Corners Problem				
Command	**Nodes Expanded**	**Memory Usage**	**Running Time (in sec)**	**Critical Analysis**
python pacman.py -l tinyCorners -p SearchAgent -a fn=bfs,prob=CornersProblem	269	$O(b^m)$ where b= branching factor of the BFS tree and m is the depth of the BFS tree.	0	Pac-Man found the path with a cost of 28
python pacman.py -l mediumCorners -p SearchAgent -a fn=bfs,prob=CornersProblem	1988		0	Pac-Man found the path with a cost of 106

Question 6 (15 points) Implement a heuristic for the CornersProblem in cornersHeuristic.

CornersHeuristic Explanation:
Here we are calculating the Manhattan distance to reach the nearest corner. After that we add the Manhattan distance to next unvisited nearest corner from the previous corner and so on and returning the sum of these Manhattan distances as corner heuristic.

Admissibility Proof: As Pac-Man is visiting the nearest corner first through Manhattan distance, and then from this corner to next nearest unvisited corner through Manhattan distance and so on till all the corners are reached. The actual cost cannot be less than this. So the heuristic is admissible.

Consistency Proof: The heuristic is consistent as the h(n)<= h(n')+c because of the usage of Manhattan distance. So when the Pacman moves to next state, the next heuristic would be always be greater than previous heuristic minus the cost because of the usage of Manhattan distance.

Corners Problem with Heuristic				
Command	**Nodes Expanded**	**Memory Usage**	**Running Time (in sec)**	**Critical Analysis**
python pacman.py -l mediumCorners -p AStarCornersAgent -z 0.5	702	O(n) where n is different positions in the grid in worst case.	0	Pac-Man found the path with a cost of 106

Question 7 (20 points) Fill in foodHeuristic in searchAgents.py with a consistent heuristic for the FoodSearchProblem.

Analysis:

The closer the value to the actual cost, the better the heuristic function. In this case, a good approximation to the distance that Pac-Man will travel from a point, is to find **the maximum** distance that Pac-Man will **at least** travel.
This can be, at any given state, the distance of the farthest food points pair. We tried by choosing **Manhattan distance** among all food points, but it didn't turn out to be such a good approximate, as its very likely that the "Manhattan path" won't cover other food points and thus will hugely underestimate the cost. We then chose **mazeDistance** as the distance parameter to find out the farthest food point pair. **mazeDistance** returns the actual path distance to reach from Node A to Node B.
To get an even closer estimate to the actual cost, we then make Pac-Man reach from its current point to any of these two points. So, we just pick the closer (Manhattan distance) of the two points found (so that we don't overestimate) and add this value to the farthest distance to get the final heuristic value.
Admissible: Both the values in the heuristic function are admissible. Pac-Man will visit the closer food point and it cannot be done in less than the Manhattan distance value. So, this

part doesn't overestimate. Also, the mazeDistance from one point to another returns the actual cost to reach that point. Therefore, this part also doesn't overestimate the cost value. Overall, both the parts do not overestimate and therefore the heuristic function is admissible.

Consistency: Most of the admissible functions are consistent. So, is the case with our heuristic function. At any point, Pac-man is never over estimating the distance value. The heuristic is consistent as h(n)<= h(n') + c because of the use of Manhattan distance and MazeDistance parameters. So, when the Pacman moves to next state, the next heuristic would be always be greater than previous heuristic minus the cost.

Food Heuristic				
Command	**Nodes Expanded**	**Memory Usage**	**Running Time (in sec)**	**Critical Analysis**
python pacman.py -l trickySearch -p AStarFoodSearchAgent	423	O(n) where n is the number of different positions in the grid.	0.1	Pac-Man found the path with a cost of 60 and a score of 570.

Q.8) Implementation of ClosestDotSearchAgent and AnyFoodSearchProblem:

- In isGoalState() of AnyFoodSearchProblem:
- x,y = *state*
  if (x,y) **in** self.foodList**:**
  self.foodList.remove(tuple([x,y]))
  return True
  else:
  return False
  Returned true if any of the food is found.
- In order to find path for the closest dot , we called the bfs on that problem.
- **def findPathToClosestDot**(self, *gameState*)**:**
- **return** search.breadthFirstSearch(problem)