**Problem 4:**

List all algorithms and methods that we have covered in this course. Write 3 sentences to describe what each algorithm and method solves etc.

Answer:

1. Depth First Search: I implemented Position Search problem instance. It's getSuccessors method returns state, action and cost. I used Stack for this problem. Stack has LIFO. list.pop() pops the last element inserted into it. A set was created to keep track of nodes visited. First I pushed start state co-ordinates and empty list of directions onto stack. The loop executed until all the nodes are popped off the fringe. Then I checked if the dequeued(popped) node's state is equal to goal state; if true I returned the list of actions for the pacman to reach its goal state. If the node is already visited, I did not push its successors onto fringe. If the node is not visited, I marked it as visited and pushed its successors onto the fringe.
2. Breadth First Search:  I implemented this problem same as DFS; only difference is it used Queue instead of Stack and Queue has FIFO property. Queue has FIFO property. list.pop() pops the last element inserted into it. Here in util.Queue() I inserted element at index 0.
3. Uniform Cost Search: The difference between the implementation of DFS and uniformCostSearch is uniformCostSearch uses PriorityQueue instead of Stack. for uniformCostSearch cost matters so we pass it as 0 for initial state and priority as 0. Here I pushed our priority as cummulative cost.
4. A\* Search: This problem also works as uniformCostSearch. It uses PriorityQueue like uniformCostSearch. One difference between uniformCostSearch and aStarSearch is we add priority as 0(cost)+heuristic of start state. another difference between uniformCostSearch and aStarSearch is we push our priority as cummulative cost + heuristic.
5. Dijkstra's algorithm: It is an algorithm that will determine the best route to take, given a number of vertices (nodes) and edges (node paths). So, if we have a graph, if we follow Dijkstra's algorithm we can efficiently figure out the shortest route no matter how large the graph is. Dijkstra's algorithm provides for us the shortest path from NodeA to NodeB. Dijkstra's algorithm can be nearly as fast as A\*, with some tweaking. A\* is generally a better implementation, but can be slightly more complex.
6. Markov Decision process: In this model, an environment is modelled as a set of states and actions can be performed to control the system’s state. The goal of this algorithm is to control the system in such a way that some performance criterium is maximized. It consists of states, actions, transitions between states and a reward function definition.
7. Minimax: In this part first Pacman is playing to maximise its own payoffs by returning the utility in case the defined depth is reached or the game is won/lost else Ghosts are playing to minimise the payoff to PacMan by calculating the next agent and increasing depth accordingly. Then, minimax action is performed for the root node by calling gameState.getLegalActions, which returns list of legal actions for an agent, and gameState.generateSuccessor, which returns the successor game state after an agent takes an action. This way it returns the minimax action from the current gameState using self.depth and self.evaluationFunction.
8. Alpha-Beta Pruning: Here first def maximizer function is implemented to check if the minimum score that the maximising player can get (beta) is higher than the highest possible score that he can obtain from another subtree (alpha) - subject to his opponent's play, and it is currently his turn. Then def minimizer function is implemented. In this function I first calculate the next agent and increase depth accordingly. Then, minimax action is performed as in Q2 for the root node but here it is implemented pacman using alpha-beta pruning by initializing alpha and beta values, minimum value that PacMan can get, and maximum value that PacMan can get.
9. Expectimax: def expectimax function is first implemented by returning the utility in case the defined depth is reached or the game is won/lost. Then I maximize for pacman and perform expectimax action for ghosts/chance nodes by calculating the next agent and increase depth accordingly. And like above implementations maximizing task for the root node i.e. pacman is performed.
10. Q-Learning:

To solve this algorithm, I implemented the update, computeValueFromQValues, getQValue, and computeActionFromQValues methods in **qlearningAgents.py**

I implemented each of these methods by following steps:

**update:**

The parent class called this to observe a state = action => nextState and reward transition. Updated Q-value as following:

Sample = R(s,a,s') + gamma\*max over actions \*Q(s'a')

            updated q value = q(s,a)=(1-alpha)q(s,a)+alpha(sample)

**computeValueFromQValues:**

For this method, I defaulted to -infinity, as 0 and negative integers can be valid values. It returned max\_action Q(state,action) where the max was over legal actions, and if there were no legal actions, which was the case at the terminal state, I returned a value of 0.0.

**getQValue:**

Here, I returned Q(state,action). It returned 0.0 if I had never seen a state or the Q node value otherwise. That implementation was done as following:

return self.values[(state, action)]

**computeActionFromQValues:**

This method computed the best action to take in a state. I got the q(s',a') max over all actions a' by this step: best\_q = self.computeValueFromQValues(state). I then checked if computed value is similar to updated value; any action from the calculated best actions is valid. If there were no legal actions, which was the case at the terminal state, it returned None.

1. Epsilon Greedy: For this question, I implemented **getAction**method in **qlearningAgents.py**. I computed the action to take in the current state. I restricted my exploration to certain level. Once I explored enough exploitation was best way. Thus, with probability self.epsilon, I took a random action and took the best policy action otherwise.