Task - 1 : Reflex Agent Evaluation Function

Introduction :

In this task we need to design an evaluation function that will empower the Reflex Agent to be able to make moves to achieve maximum utility while considering the locations of both food pellets and ghosts.

Problem Analysis:

In this task, we need to design an evaluation function with the help of (state, action). In pacman.py the extractable information from the game state is ***successorGameState, newPos, newFood, newGhostStates***. Here :

- successorGameState is the modified game configuration after performing the particular action
- newPos is the position of the reflex agent after performing the action
- newFood is the food pellets that remain after performing the action
- newGhostState is the state of the ghosts after performing the action.

An evaluation function basically provides a numerical measurement of how desirable a specific state is. Here maximized value represents the desirability. So our evaluation function needs to maximize this value by returning a large number if the successorGameState is a win state considering a particular action.

Solution Explanation:

I designed the evaluation function using 3 factors. These are the Proximity of the food pellet, the Proximity of the ghosts, and the Ghost Threat Level.

Proximity of the Food Pellet: Closer food means a better score since it is optimal for Pacman to eat nearby food rather than staying idle for a while. So I calculated the minimum Manhattan distance between Pacman and the nearest food pellet as Food Proximity.

Proximity of the ghost: Since Pacman wants ghosts to be far away from him to decrease the risk of being captured, I considered and computed the minimum Manhattan distance between Ghost and Pacman as part of the evaluation function.

Ghost Threat Level : If ghosts are too close to Pacman, we are considering this as a threat to the pacman.

```
if minGhostDistance < 2:
    return -math.inf
```

Here we measured the threat level using a certain range as shown above. We are returning a negative infinity score to avoid capturing of the Pacman by ghosts

And finally, the score will be calculated by adding the inverse of minFoodDistance and subtracting the inverse of the minGhostDistance with the current score. This helps is prioritizing closer food and keeping ghosts further away.

Behavior in case of different hyperparameters :

- Taking Euclidean distance instead of Manhattan distance while calculating distance fails in some test cases since here we are not considering walls

Challenges :

Faced some challenges initially while deciding about the factors of the evaluation function (Food Distance, Ghost Distance etc). Tinkered other probable factors and finally came up with this evaluation function.

Task - 2 : Minimax

Introduction :

We had to implement a minimax agent in and find out minimax actions in this particular problem. Here each agents play optimally and the gameplay is deterministic

Problem Analysis :

In the given scenario, Pacman tries to maximize the score and ghosts try to minimize the score. So a minimax tree is constructed in the problem where each level alternates between Pacman move and ghost move. The pacman move is the maximizer and the ghost move is the minimizer. This minimax tree represents all possible moves of the agents and their respective outcomes.

Here we need two separate functions for the maximizer and the minimizer. Since Pacman wants to maximize the value, the maximizer function "***maxValue()***" will be invoked by Pacman. On the other hand, the minimizer function "***minValue()***" will be invoked by the ghost. The order of the moves is determined by the ***agentIndex (0 for pacman and 1 or higher for the ghosts)*** and ***depth*** of the minimax tree.

Solution Explanation:

Since the number of agents is not fixed, we utilized a recursion strategy to handle any number of Pacman and Ghosts. The value() function does the work of selecting minimization or maximization function based on the current agent's role (By Agent Index) - (minValue() for Ghosts and maxValue() for Pacaman)

```
def value(self, gameState, agentIndex, depth):
    if gameState.isWin() or gameState.isLose() or depth == 0:
        return self.evaluationFunction(gameState), Directions.STOP
    elif agentIndex == 0:
        return self.maxValue(gameState, agentIndex, depth)
    elif agentIndex > 0:
        return self.minValue(gameState, agentIndex, depth)
```

Initially, it checks the terminal conditions (win, lose, depth). Then it alternates between maximizer and minimizer. The value() method is recursively called for each legal action to

evaluate the resulting state. The method updates the best score and action if a higher score is found by backtracking. Eventually, it returns the score and action which guided toward optimal gameplay.

Challenges :

No challenges were faced in this problem as this was demonstrated in the lab.

Task - 3 : Alpha-Beta Pruning

Introduction :

We had to implement an alpha-beta agent in this particular problem. Alpha-Beta agent avoids exploring the less important or unnecessary branches of the minimax tree which saves the computation time.

Problem Analysis :

Here the problem is similar to the previous minimax problem but here we are optimizing our tree search by pruning or avoiding the unnecessary branches of the tree. This pruning is called alpha-beta pruning.

In a minimax tree, when evaluating a node we look at its parent. If the parent is a maximizer it expects a higher score. Contrary, if the parent is a minimizer node, it expects a lower score.

In alpha-beta pruning, if a minimizer node finds a value that has a value lower than the current maximum, it won't explore that branch afterward. Similarly, if a maximizer node finds a node that has a value higher than the current minimum, it stops exploring. So Alpha Beta pruning reduces the computational cost of traversing whole trees by pruning the less important subtrees. It still ensures the optimal result at the root node.

Solution Explanation :

There are some similarities with the previous minimax solution. But here two additional parameters are introduced - Alpha and Beta. These are used to keep track of the best values found so far in the current branch. Alpha denotes the highest value encountered by the Pacman and Beta denotes the lowest value encountered by the ghosts.

Alpha and Beta are initialized to negative infinity and positive infinity respectively. These extreme values are taken so that any encountered value can get updated accordingly during the minimax search.

In the ***minValue()*** function we are now comparing currValue with the alpha and in the ***maxValue()*** function, we are comparing currValue with the beta. If the algorithm finds a value greater than alpha for Pacman or less than beta for ghosts, it updates alpha/beta accordingly. And finally returning the value and the action which leads to pruning or avoiding the less important branches.

Challenges :

No challenges were faced as this was quite similar to the previous one.

Task - 4 : Expectimax

Introduction :

We had to implement an expectimax agent that utilizes the probability to make suboptimal decisions of actions in this problem.

Problem Analysis :

In expectimax search tree, we have a stochastic agent in the place of a minimizer. So the decisions will be made depending upon probabilities. In this particular problem, we are assuming that probability of each move is equally likely. So we compute the probability as : 1/Number of legal moves. Here we don't compare the score with a reference value. Rather, we compute the expected score by continuously adding the product of the probability (p) and the value obtained from the helper function with each move.

Solution Explanation :

In expectimax, we are dealing with agents (like Ghosts) who don't always make decisions in the deterministic process. We model this uncertainty by assuming ghosts choose moves randomly from their legal options.
Here Pacman's goal remains the same so the maximizer is similar to the previous minimax trees.
But to handle the uncertainty of ghost moves, we calculate the probability of each legal action for the ghosts.
P = 1/Number of legal moves
Then we need to compute the expected value of all possible legal action for that probability.

```
    for action in legalActions:
        successorGameState = gameState.generateSuccessor(agentIndex, action)
        successorScore, _ = self.value(successorGameState, nextAgent, nextDepth)

        currValue += prob * successorScore
        currAction = action
```

This expected value ***(currValue)*** represents the average utility that Pacman can achieve, considering the unpredictable behavior of the ghosts

Challenges :

No challenges were faced as this was quite similar to the previous one.

Task - 5 : Better Evaluation Function

Introduction :

In this task, we need to design a better evaluation function for evaluating the current state.

Problem Analysis:

In this task we need to design an evaluation function with the help of (state, action). In pacman.py the extractable information from the game state is ***successorGameState, newPos, newFood, newGhostStates***. Here :

- successorGameState is the modified game configuration after performing the particular action
- newPos is the position of the reflex agent after performing the action
- newFood is the food pellets that remain after performing the action
- newGhostState is the state of the ghosts after performing the action.

An evaluation function basically provides a numerical measurement of how desirable a specific state is. Having a good evaluation function significantly reduces the number of branches we need to explore in a tree thus reducing computational cost.

So we need to implement a better evaluation function, unlike the previous task. We have to consider only the state here.

Solution Explanation :

We needed to take some extra parameter like foodValue, ghostValue, and scaredGhostValue to implement this evaluation function.

We calculate the minimum Manhattan distance for food distance, multiply the reciprocal by foodValue, and then add the score. The pacman score will benefit from the smallest possible food distance because not consuming food pallets would lower the score.

We also took into account the ghosts' level of fear when calculating the distance to them. We increase the score's reward during the times when ghosts are most vulnerable. As a result, in these circumstances, we set ScaredGhostValue to a higher value of 100. However, we offer a lower reward, set at 10, in normal circumstances when ghosts are not afraid.

We calculated the score based on “if the ghost is scared or not” like below :

```
for ghost in newGhostStates:
    ghostDistance = manhattanDistance(newPos, ghost.getPosition())
    if ghostDistance > 0:
        if ghost.scaredTimer > 2:
            score += scaredGhostValue*(1/ghostDistance)
        else:
            score -= ghostValue*(1/ghostDistance)
    else:
        return -math.inf
```

Finally returned the score from the evaluation function.

Challenges:

Faced challenges while deciding about extra parameters and their values as Achieving the desired gameplay outcomes requires finding the ideal balance between these hyperparameters.