Lab 4 - Multi-Agent Search

CSE 4712: Artificial Intelligence Lab

Submitted by-

ID: 190041220

Name: Tasfia Tasneem Annesha

Dept: Computer Science and Engineering

Group: 2B

Task 1:

Explanation:

To enhance the ReflexAgent's performance, I focused on improving the evaluationFunction() as follows:First, I started by figuring out how far away each ghost was from Pacman by calculating the Manhattan distance from Pacman to each ghost. If this distance equals zero, meaning Pacman and a ghost are in the same position, I checked whether the ghost's "scared" timer is greater than zero.The ghost would catch Pacman and score -1x10^9 if it was no longer afraid. In the event that the ghost remains scared, Pacman would receive a score of 1x10^9, which indicates a favorable circumstance.I examined the available foods when neither of these scenarios applied. Using a loop, I computed the Manhattan distance between Pacman and each food item and determined the closest food item by selecting the minimum distance. The score returned is the reciprocal of the closest food distance, with 1 added to the denominator to prevent division by zero.The goal of this tactic is to help Pacman find the nearest food source while avoiding threatening ghosts and using the scared timer to guide his decisions.

Code:

```
class ReflexAgent(Agent):

    def getAction(self, gameState):

        # Collect legal moves and successor states
        legalMoves = gameState.getLegalActions()

        # Choose one of the best actions
        scores = [self.evaluationFunction(gameState, action) for action in legalMoves]
        bestScore = max(scores)
        bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
        chosenIndex = random.choice(bestIndices) # Pick randomly among the best

        "Add more of your code here if you want to"
        return legalMoves[chosenIndex]

    def evaluationFunction(self, currentGameState, action):
        """

        # Useful information you can extract from a GameState
(pacman.py)
        successorGameState =
currentGameState.generatePacmanSuccessor(action)
        newPos = successorGameState.getPacmanPosition()
        newFood = successorGameState.getFood()
        newGhostStates = successorGameState.getGhostStates()
        newScaredTimes = [ghostState.scaredTimer for ghostState in
newGhostStates]

        "*** YOUR CODE HERE ***"
        for ghost in newGhostStates:
            ghostDistancefromPacman =
util.manhattanDistance(newPos,ghost.getPosition())
            if ghostDistancefromPacman==0:
                if ghost.scaredTimer>0:
                    return 1e9
                elif ghost.scaredTimer==0:
                    return -1e9
            closestfood = 1e9
            foods= currentGameState. getFood().asList()
            for food in foods:
                distance = util.manhattanDistance(newPos,food)
                closestfood = min(distance,closestfood)

            return 1/(closestfood+1)


        #return successorGameState.getScore()
```

Task 2:

Explanation:

This task involved implementing a Minimax agent with three main functions:

The value function takes the game state, agent index, and depth, returning an action by calling either the maximizer or minimizer function based on the agent.The maximizer function, used for Pacman, calculates the maximum score among successor states as long as the depth hasn't been reached and the game is not in a win/lose state.The minimizer function, called for all ghosts, computes the minimum score among successor states, considering the game's current state.

Code:

```
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (question 2)
    """

    def getAction(self, gameState):
                "*** YOUR CODE HERE ***"
        # util.raiseNotDefined()
        return self.value(gameState, 0, self.depth)[1]

    def value(self, gameState, agentIndex, depth):
        if gameState.isLose() or gameState.isWin() or depth == 0:
            return self.evaluationFunction(gameState), Directions.STOP
        elif agentIndex == 0:
            return self.maximizer(gameState, agentIndex, depth)
        else:
            return self.minimizer(gameState, agentIndex, depth)

    def minimizer(self, gameState, agentIndex, depth):
        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        if nextAgent == 0:
            nextDepth = depth - 1
        else:
            nextDepth = depth

        currentScore, currentAction = 1e9, Directions.STOP
        actionList = gameState.getLegalActions(agentIndex)

        for action in actionList:
            successorGameState =
gameState.generateSuccessor(agentIndex, action)
```

```
            successorScore = self.value(successorGameState, nextAgent, 
nextDepth)[0]
            if currentScore > successorScore:
                currentScore = successorScore
                currentAction = action

        return currentScore, currentAction

    def maximizer(self, gameState, agentIndex, depth):
        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        if nextAgent == 0:
            nextDepth = depth - 1
        else:
            nextDepth = depth

        currentScore, currentAction = -1e9, Directions.STOP
        actionList = gameState.getLegalActions(agentIndex)

        for action in actionList:
            successorGameState = 
gameState.generateSuccessor(agentIndex, action)
            successorScore = self.value(successorGameState, nextAgent, 
nextDepth)[0]
            if currentScore < successorScore:
                currentScore = successorScore
                currentAction = action

        return currentScore, currentAction
```

Task 3:

Explanation:

In this task,I implemented an alpha-beta pruning agent, following the provided code. It's quite similar to the minimax agent, with a few key differences:

The value function now has additional parameters, alpha and beta.The maximizer function takes alpha and beta as new parameters. It returns the maximum score from successors, but if the score exceeds beta, it's returned. Otherwise, it returns the maximum between alpha and the selected score.

Similarly, the minimizer function takes alpha and beta as extra parameters. It returns the minimum score from successors, and if the score is smaller than alpha, it's returned. Otherwise, it returns the minimum between the selected score and beta.

Code:

```
class AlphaBetaAgent(MultiAgentSearchAgent):
    """
    Your minimax agent with alpha-beta pruning (question 3)
    """

    def getAction(self, gameState):
        """
        Returns the minimax action using self.depth and self.evaluationFunction
        """
        "*** YOUR CODE HERE ***"
        # util.raiseNotDefined()
        return self.value(gameState, 0, self.depth, -1e9, 1e9)[1]

    def value(self, gameState, agentIndex, depth, alpha, beta):
        if gameState.isWin() or gameState.isLose() or depth == 0:
            return self.evaluationFunction(gameState), Directions.STOP
        elif agentIndex == 0:
            return self.maximizer(gameState, agentIndex, depth, alpha, beta)
        else:
            return self.minimizer(gameState, agentIndex, depth, alpha, beta)

    def minimizer(self, gameState, agentIndex, depth, alpha, beta):
        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        if nextAgent == 0:
            nextDepth = depth - 1
        else:
            nextDepth = depth

        actionList = gameState.getLegalActions(agentIndex)
        currentScore, currentAction = 1e9, Directions.STOP
        for action in actionList:
            successorGameState =
gameState.generateSuccessor(agentIndex, action)
            successorScore = self.value(successorGameState, nextAgent,
nextDepth, alpha, beta)[0]

            if successorScore < currentScore:
                currentScore = successorScore
                currentAction = action

            if currentScore < alpha:
                return currentScore, currentAction
            beta = min(beta, currentScore)

        return currentScore, currentAction

    def maximizer(self, gameState, agentIndex, depth, alpha, beta):
        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        if nextAgent == 0:
            nextDepth = depth - 1
        else:
            nextDepth = depth

        actionList = gameState.getLegalActions(agentIndex)
        currentScore, currentAction = -1e9, Directions.STOP

        for action in actionList:
            successorGameState =
gameState.generateSuccessor(agentIndex, action)
            successorScore = self.value(successorGameState, nextAgent,
nextDepth, alpha, beta)[0]

            if successorScore > currentScore:
                currentScore = successorScore
                currentAction = action

            if currentScore > beta:
                return currentScore, currentAction
            alpha = max(alpha, currentScore)

        return currentScore, currentAction
```

Task 4:

Explanation:
We were tasked with implementing an expectimax agent in this assignment, which is identical to a previous task. The dispatcher and maximizer operations are not affected. The minimizer function, on the other hand, is replaced by an "expected value calculator" function.The selected score is calculated by estimating the expected value of the score over all feasible sets of activities. This is achieved by taking a weighted sum of the scores for each action, where the weight corresponds to the likelihood of choosing that action. The function eventually returns the expected value as well as a randomly selected action.

Code:

```
class ExpectimaxAgent(MultiAgentSearchAgent):

    def getAction(self, gameState):

        "*** YOUR CODE HERE ***"
        # util.raiseNotDefined()
        return self.value(gameState, 0, self.depth)[1]


    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)
        else:
            return self.expValue(gameState, agentIndex, depth)

    def expValue(self, gameState, agentIndex, depth):
        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        if nextAgent == 0:
            nextDepth = depth - 1
        else:
            nextDepth = depth

        actionList = gameState.getLegalActions(agentIndex)
        expectedValue, prob = 0, 0
        for action in actionList:
            successorGameState = gameState.generateSuccessor(agentIndex, action)
            successorScore = self.value(successorGameState, nextAgent, nextDepth)[0]

            prob = 1/len(actionList)
            expectedValue = expectedValue + prob * successorScore

        return expectedValue, random.choice(actionList)


    def maxValue(self, gameState, agentIndex, depth):
        nextAgent = (agentIndex + 1) % gameState.getNumAgents()
        if nextAgent == 0:
            nextDepth = depth - 1
        else:
            nextDepth = depth

        actionList = gameState.getLegalActions(agentIndex)
        selectedScore, selectedAction = -1e9, Directions.STOP

        for action in actionList:
            successorGameState = gameState.generateSuccessor(agentIndex, action)
            successorScore = self.value(successorGameState, nextAgent, nextDepth)[0]

            if successorScore > selectedScore:
                selectedScore = successorScore
                selectedAction = action

        return selectedScore, selectedAction
```

Task 5:

Explanation:

In this task, I improved the evaluation function, focusing on Pacman's interaction with ghosts. I considered the Manhattan distance between Pacman and each ghost. If the distance was less than or equal to 1, and the ghost wasn't scared, a score of -1x10^9 was returned, indicating Pacman's loss. Otherwise, a score of 1x10^9 was awarded.

I calculated the closest food's Manhattan distance to Pacman and also the nearest capsule's distance using loops. The new score was determined as 1 divided by (closest food distance + 1)+1 divided by (closest capsule distance + 1)+ the current score, with the addition of 1 to avoid division by zero. This new score was returned to evaluate the game state.

Code:

```
def betterEvaluationFunction(currentGameState):

    "*** YOUR CODE HERE ***"
    #util.raiseNotDefined()
    currPos = currentGameState.getPacmanPosition()
    currGhostStates = currentGameState.getGhostStates()

    for ghost in currGhostStates:
        ghostDistance = util.manhattanDistance(currPos,
ghost.getPosition())
        if ghostDistance<=1:
            if ghost.scaredTimer>0:
                return 1e9
            elif ghost.scaredTimer==0:
                return -1e9

    currScore = currentGameState.getScore()

    closestFoodDist = 1e9
    foods2 = currentGameState.getFood().asList()
    for food2 in foods2:
        dist2 = util.manhattanDistance(currPos, food2)
        closestFoodDist = min(dist2,closestFoodDist)

    closestCapDist = 1e9
    caps = currentGameState.getCapsules()
    for cap in caps:
        dist2 = util.manhattanDistance(currPos, cap)
        closestCapDist = min(dist2,closestCapDist)
    NewScore = (1/(closestFoodDist + 1)) +  1/(closestCapDist + 1) + currScore

    return  NewScore
```