Homework 3: Multi-Agent Searh

Part I. Implementation

Part1. Minimax

```
        # Begin your code (Part 1)
        """
        In the recursion of minimax.
        When it's pacman turn, we choose the max value.
        When it's ghosts turn, we choose the min value.
        When it's depth = 0 or win or Lose, we evaluate the state value
        Then, we choose the action with biggest value among all of the returns
        """
        legalActions = getNonStopActions(gameState, 0)
        values = [self.minimax(gameState.getNextState(0, action), self.depth, 1) for action in legalActions]
        maxi = max(values)
        max_indexs = [i for i in range(len(legalActions)) if (values[i] == maxi)]

        choose = random.choice(max_indexs)
        return legalActions[max_indexs[0]]
        # End your code (Part 1)

    def minimax(self, state, depth, agentIndex):
        if depth == 0 or state.isWin() or state.isLose():
            return self.evaluationFunction(state)

        # Pacman's turn (maximizing player)
        if agentIndex == 0:
            return max(self.minimax(state.getNextState(agentIndex, action), depth, 1) for action in
                       getNonStopActions(state, agentIndex))
        # Ghosts' turn (minimizing players)
        else:
            nextAgent = agentIndex + 1
            if nextAgent == state.getNumAgents():
                nextAgent = 0  # Reset to Pacman's turn
            if nextAgent == 0:
                depth -= 1  # Reduce depth when it's Pacman's turn again
            return min(self.minimax(state.getNextState(agentIndex, action), depth, nextAgent) for action in
                       getNonStopActions(state, agentIndex))
```

Part2. AlphaBetaAgent

```
        # Begin your code (Part 2)
        """
        The recursion of alpha_beta_pruning is like minimax function.
        However, we prune the possible outcomes to speed up the recursion.
        When we find a max value in finding max turn,
            we set it to be a lower bound (alpha) of recursion.
            That is, if we have a value which definitely lower than the lower bound (beta < alpha)
            the outcome will not be our choose, so we break the recursion (pruing)
        Similarly, we set upper bound (beta) in finding min turn
        Finally, we choose the action with biggest value among all of the returns
        """
        legalActions = getNonStopActions(gameState, 0)
        values = []
        bestAction = None
        bestValue = float('-inf')
        alpha = float('-inf')
        beta = float('inf')
        for action in legalActions:
            value = self.alpha_beta_pruning(gameState.getNextState(0, action), self.depth, 1, alpha, beta)
            if value > bestValue:
                bestValue = value
                bestAction = action
            alpha = max(alpha, bestValue)
            values.append(value)

        maxi = max(values)
        max_indexs = [i for i in range(len(legalActions)) if (values[i] == maxi)]
        choose = random.choice(max_indexs)
        return legalActions[max_indexs[0]]
        # End your code (Part 2)

    def alpha_beta_pruning(self, state, depth, agentIndex, alpha, beta):
        if depth == 0 or state.isWin() or state.isLose():
            return self.evaluationFunction(state)

        # Pacman's turn (maximizing player)
        if agentIndex == 0:
            v = float('-inf')
            for action in getNonStopActions(state, agentIndex):
                v = max(v, self.alpha_beta_pruning(state.getNextState(agentIndex, action), depth, 1, alpha, beta))
                alpha = max(alpha, v)
                if beta < alpha:
                    break
            return v

        # Ghosts' turn (minimizing players)
        else:
            nextAgent = agentIndex + 1
            if nextAgent == state.getNumAgents():
                nextAgent = 0  # Reset to Pacman's turn
            if nextAgent == 0:
                depth -= 1     # Reduce depth when it's Pacman's turn again

            v = float('inf')
            for action in getNonStopActions(state, agentIndex):
                v = min(v, self.alpha_beta_pruning(state.getNextState(agentIndex, action), depth, nextAgent, alpha, beta))
                beta = min(v, beta)
                if beta < alpha:
                    break
            return v
```

Part3. Expectimax

```
        # Begin your code (Part 3)
        """
        In the recursion of minimax.
        When it's pacman turn, we choose the max value.
        When it's ghosts turn, we take weighted sum of values.
        ( In this case, we take uniform of all values )
        When it's depth = 0 or win or Lose, we evaluate the state value
        Then, we choose the action with biggest value among all of the returns
        """
        legalActions = getNonStopActions(gameState, 0)
        bestAction = None
        bestValue = float('-inf')
        values = []

        for action in legalActions:
            v = self.expectimax(gameState.getNextState(0, action), self.depth, 1)
            if v > bestValue:
                bestAction = action
                bestValue = v
            values.append(v)

        maxi = max(values)
        max_indexs = [i for i in range(len(legalActions)) if (values[i] == maxi)]
        choose = random.choice(max_indexs)
        return legalActions[max_indexs[0]]
        # End your code (Part 3)

    def expectimax(self, state, depth, agentIndex):

        if depth == 0 or state.isWin() or state.isLose():
            return self.evaluationFunction(state)

        if agentIndex == 0:  # Pacman's turn (maximizing player)
            return max(self.expectimax(state.getNextState(agentIndex, action), depth, 1) for action in
                       getNonStopActions(state, agentIndex))

        else:  # Ghosts' turn (chance node)
            nextAgent = agentIndex + 1
            if nextAgent == state.getNumAgents():
                nextAgent = 0  # Reset to Pacman's turn
            if nextAgent == 0:
                depth -= 1     # Reduce depth when it's Pacman's turn again

            legalActions = getNonStopActions(state, agentIndex)
            numActions = len(legalActions)
            return sum(self.expectimax(state.getNextState(agentIndex, action), depth, nextAgent) for action in
                       legalActions) / numActions
```

Part4. betterEvaluationFunction

```
# Begin your code (Part 4)
"""
We evaluate state based on NonScared ghosts, Scared ghost, Food, and Capsule
First, if lose or almost loss ( minNonScared <= 1)
    we return -300000 or -200000
Second, if we could eat the Scared Ghost (minScaredTime > minScared)
    evaluation += 150000 * (1 / minScared)
Third, we encourage pacman to eat capsules,
    evaluatoin += 200 * (1 / nearCapsule) + 10
Fourth, we encourage pacman to eat foods,
    evaluation += 10 * (1 / nearFood) + 5
Besides, we use bfs to find the correct distance to object,
    and we initialize evaluation = 2 * score to let it sensitive to the change of score
"""
dxs = [1, 0, -1, 0]
dys = [0, 1, 0, -1]

def findEnd(start, end):
    q = Queue()
    q.push(start)
    dist = {}
    dist[start] = 0
    while not q.isEmpty():
        xy = q.pop()
        if xy == end:
            return dist[xy]
        for dx, dy in zip(dxs, dys):
            if not currentGameState.hasWall(xy[0] + dx, xy[1] + dy) and (xy[0] + dx, xy[1] + dy) not in dist:
                q.push((xy[0] + dx, xy[1] + dy))
                dist[(xy[0] + dx, xy[1] + dy)] = dist[xy] + 1
    return None

def findFood(start):
    q = Queue()
    q.push(start)
    dist = {}
    dist[start] = 0
    while not q.isEmpty():
        xy = q.pop()
        if currentGameState.hasFood(xy[0], xy[1]):
            return dist[xy]
        for dx, dy in zip(dxs, dys):
            if not currentGameState.hasWall(xy[0] + dx, xy[1] + dy) and (xy[0] + dx, xy[1] + dy) not in dist:
                q.push((xy[0] + dx, xy[1] + dy))
                dist[(xy[0] + dx, xy[1] + dy)] = dist[xy] + 1
    return None

# score, pos
score = currentGameState.getScore()
pos = currentGameState.getPacmanPosition()
# if currentGameState.isLose():
#     return -10000

# food
foods = currentGameState.getFood().asList()
nearFood = findFood(pos)

# capsule
capsule = [findEnd(pos, capsule_pos) for capsule_pos in currentGameState.getCapsules()
           if findEnd(pos, capsule_pos) is not None]
nearCapsule = min(capsule) if len(capsule) > 0 else 9999
```

```
    # ghosts
    ghosts = currentGameState.getGhostStates()
    minScared = 9999
    minNonScared = 9999
    minScaredTime = 9999
    for ghost in ghosts:
        dist = findEnd(pos, ghost.getPosition())
        if ghost.scaredTimer > 0 and dist is not None:
            minScared = min(minScared, dist)
            minScaredTime = ghost.scaredTimer
        elif ghost.scaredTimer == 0 and dist is not None:
            minNonScared = min(minNonScared, dist)

    # main evaluation
    evaluation = 2 * score
    if currentGameState.isLose():
        return -300000
    if minNonScared <= 1:
        return -200000
    if minScaredTime > minScared:
        evaluation += 150000 * (1 / minScared)

    if len(capsule) > 0:
        evaluation += 200 * (1 / nearCapsule) + 10

    if nearFood is not None:
        evaluation += 10 * (1 / nearFood) + 5

    # print(f"near food = {nearFood}, evaluation = {evaluation}")
    return evaluation
```

Part II. Results & Analysis

Part1. Minimax

```
*** PASS: test_cases\part1\4-two-ghosts-3level.test
*** PASS: test_cases\part1\5-two-ghosts-4level.test
*** PASS: test_cases\part1\6-tied-root.test
*** PASS: test_cases\part1\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\part1\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\part1\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\part1\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\part1\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\part1\7-2c-check-depth-two-ghosts.test
*** Running MinimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running MinimaxAgent on smallClassic after 1 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part1\8-pacman-game.test

### Question part1: 15/15 ###
```

Part2. AlphaBetaAgent

```
*** PASS: test_cases\part2\2-4b-vary-depth.test
*** PASS: test_cases\part2\2-one-ghost-3level.test
*** PASS: test_cases\part2\3-one-ghost-4level.test
*** PASS: test_cases\part2\4-two-ghosts-3level.test
*** PASS: test_cases\part2\5-two-ghosts-4level.test
*** PASS: test_cases\part2\6-tied-root.test
*** PASS: test_cases\part2\7-1a-check-depth-one-ghost.test
*** PASS: test_cases\part2\7-1b-check-depth-one-ghost.test
*** PASS: test_cases\part2\7-1c-check-depth-one-ghost.test
*** PASS: test_cases\part2\7-2a-check-depth-two-ghosts.test
*** PASS: test_cases\part2\7-2b-check-depth-two-ghosts.test
*** PASS: test_cases\part2\7-2c-check-depth-two-ghosts.test
*** Running AlphaBetaAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running AlphaBetaAgent on smallClassic after 1 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part2\8-pacman-game.test
```

part3. Expectimax

```
*** PASS: test_cases\part3\6-1b-check-depth-one-ghost.test
*** PASS: test_cases\part3\6-1c-check-depth-one-ghost.test
*** PASS: test_cases\part3\6-2a-check-depth-two-ghosts.test
*** PASS: test_cases\part3\6-2b-check-depth-two-ghosts.test
*** PASS: test_cases\part3\6-2c-check-depth-two-ghosts.test
*** Running ExpectimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running ExpectimaxAgent on smallClassic after 1 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part3\7-pacman-game.test

### Question part3: 20/20 ###
```

Part4. betterEvaluationFunction

```
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\part4\grade-agent.test (8 of 8 points)
*** EXTRA CREDIT: 2 points
***     1291.7 average score (4 of 4 points)
***         Grading scheme:
***          < 600:  0 points
***         >= 600:  2 points
***         >= 1200:  4 points
***     10 games not timed out (2 of 2 points)
***         Grading scheme:
***          < 0:  fail
***         >= 0:  0 points
***         >= 5:  1 points
***         >= 10:  2 points
***     10 wins (4 of 4 points)
***         Grading scheme:
***          < 1:  fail
***         >= 1:  1 points
***         >= 4:  2 points
***         >= 7:  3 points
***         >= 10:  4 points

### Question part4: 10/10 ###
```

Epligogue

```
Finished at 16:36:12

Provisional grades
==================
Question part1: 15/15
Question part2: 20/20
Question part3: 20/20
Question part4: 10/10
------------------
Total: 65/65
```