Homework 3: Multi-Agent Search

111550088 張育維

Part I. Implementation (20%):

```
# Begin your code (Part 1)
"""
In part 1, I define a function called 'value' for recursion. I use 'gameState.getLegalAction
(index)' to get a list of this agent's legal moves, and then use a for loop to call the
'value' function recursively. I will use 'getNextState' to convert these legal moves into game
states. In the 'value' function, the recursion stops when the current state indicates a win or
a loss, or when the specified recursion depth is reached (determined by 'self.depth'). When
recursion stops, the function calls 'evaluationFunction(state)' to get the final score and
return it. At each depth, every agent moves once, so if the index of the current agent equals
the starting index (determined by 'self.index'), the depth increases by one. Each agent
retrieves all their legal moves in the same way and recursively calls the 'value' function. If
the index of the agent is zero, then the function returns the maximum value among all child
states; if the index is not zero, it returns the minimum value.
"""
def value(state, dep, idx):
    if state.isWin() or state.isLose():
        return self.evaluationFunction(state)
    idx = idx + 1
    if idx == gameState.getNumAgents():
        idx = 0
    if idx == self.index:
        dep = dep + 1
    if dep > self.depth:
        return self.evaluationFunction(state)
    list = state.getLegalActions(idx)
    v = 0
    if idx == 0:
        v = float('-inf')
    else:
        v = float('inf')
    for a in list:
        if idx == 0:
            v = max(v, value(state.getNextState(idx, a), dep, idx))
        else:
            v = min(v, value(state.getNextState(idx, a), dep, idx))
    return v
```

```
index = self.index
action = gameState.getLegalActions(index)
re = action[index]
v = float('-inf')
for a in action:
    v1 = value(gameState.getNextState(index, a), 1, index)
    if index == 0:
        if v < v1:
            re = a
            v = v1
    else:
        if v > v1:
            re = a
            v = v1
return re
# raise NotImplementedError("To be implemented")
# End your code (Part 1)
```

```
Your minimax agent with alpha-beta pruning (Part 2)
"""
"""
In this part, most of the logic is similar to part 1. However, in the 'value' function, I add two
components called alpha and beta for alpha-beta pruning. When the index equals zero (where we
select the maximum), if the return value is greater than alpha, I update the alpha value. If the
return value is greater than beta, I stop the recursion and return the value immediately. When the
index is not zero (where we select the minimum), if the return value is smaller than beta, I
update the beta value. If the return value is smaller than alpha, I stop the recursion and return
the value immediately. The initial values for alpha and beta are negative infinity and positive
infinity, respectively. The for-loop for the initial state's legal actions also uses the alpha and
beta values to determine whether to continue recursion or to prune the remaining branches.
"""
def getAction(self, gameState):
    """
    Returns the minimax action using self.depth and self.evaluationFunction
    """
    # Begin your code (Part 2)
    def value(state, dep, idx, alpha, beta):
        if state.isWin() or state.isLose():
            return self.evaluationFunction(state)
        idx = idx + 1
        if idx == gameState.getNumAgents():
            idx = 0
        if idx == self.index:
            dep = dep + 1
        if dep > self.depth:
            return self.evaluationFunction(state)
        list = state.getLegalActions(idx)
        v = 0
        if idx == 0:
            v = float('-inf')
        else:
            v = float('inf')
        for a in list:
            if idx == 0:
```

```
    for a in list:
        if idx == 0:
            temp = value(state.getNextState(idx, a), dep, idx, alpha, beta)
            v= max(v, temp)
            if v > beta:
                return v
            alpha = max(alpha, v)
        else:
            temp = value(state.getNextState(idx, a), dep, idx, alpha, beta)
            v = min(v, temp)
            if v < alpha:
                return v
            beta = min(beta, v)
    return v
index = self.index
action = gameState.getLegalActions(index)
re = action[index]
v = float('-inf')
alpha = float('-inf')
beta = float('inf')
for a in action:
    v1 = value(gameState.getNextState(index, a), 1, index, alpha, beta)
    if index == 0:
        if v < v1:
            re = a
        v = max(v, v1)
        if v > beta:
            return re
        alpha = max(alpha, v)
    else:
        if v > v1:
            re = a
        v = min(v1, v)
        if v < alpha:
            return re
        beta = min(beta, v)
return re
```

```
  Your expectimax agent (Part 3)
"""
"""
This part is similar to part 1. The difference is that, originally, when the index is not zero, we
select the minimum value. In this part, instead of selecting the minimum, we calculate the average
value of all legal actions. However, for the initial state's legal actions, if the index of the
initial state is not zero, I still choose the minimum value instead of the average because it must
return an action.
"""
def getAction(self, gameState):
    """
    Returns the expectimax action using self.depth and self.evaluationFunction

    All ghosts should be modeled as choosing uniformly at random from their
    legal moves.
    """
    # Begin your code (Part 3)
    def value(state, dep, idx):
        if state.isWin() or state.isLose():
            return self.evaluationFunction(state)
        idx = idx + 1
        if idx == gameState.getNumAgents():
            idx = 0
        if idx == self.index:
            dep = dep + 1
        if dep > self.depth:
            return self.evaluationFunction(state)
        list = state.getLegalActions(idx)
        v = 0
        sum = 0
        if idx == 0:
            v = float('-inf')
        else:
            v = float('inf')
        for a in list:
            if idx == 0:
                v = max(v, value(state.getNextState(idx, a), dep, idx))
```

```
            else:
                sum = sum + value(state.getNextState(idx, a), dep, idx)
        if idx != 0:
            v = sum / len(list)
        return v

    index = self.index
    action = gameState.getLegalActions(index)
    re = action[index]
    v = float('-inf')
    for a in action:
        v1 = value(gameState.getNextState(index, a), 1, index)
        if index == 0:
            if v < v1:
                re = a
                v = v1
        else:
            if v > v1:
                re = a
                v = v1
    return re

    # raise NotImplementedError("To be implemented")
    # End your code (Part 3)
```

```
evaluation function (Part 4).
"""
"""
There are three components in my evaluation function: 'score', 'fscore', and 'gscore'. I obtain
'score' directly from the function 'getScore()'. 'fscore' is calculated based on the distance to
the nearest food pellet. The formula is '10 / distance + 8', since a shorter distance to food
indicates a higher score potential. 'gscore' is calculated based on the scared time and the
distance to the nearest ghost. If the scared time is greater than 0, 'gscore' increases by '300 /
distance' if the scared time is over 10, or '150 / distance' if the scared time is 10 or less. If
the scared time is 0, 'gscore' decreases by '-15 / distance'. This approach reflects that Pacman
should capture ghosts when they're scared and avoid them when they're not. The final score is the
sum of 'score', 'fscore', and 'gscore'.
"""
# Begin your code (Part 4)
position = currentGameState.getPacmanPosition()     #get the position now
food = currentGameState.getFood()       #eat the point
ghosts = currentGameState.getGhostStates()      #a list
scaredtimes = [state.scaredTimer for state in ghosts]
score = currentGameState.getScore()
nearestdist = min([manhattanDistance(position, s.getPosition()) for s in ghosts])

food_dist = 0
if(len(food.asList()) > 0):
    food_dist = min([manhattanDistance(position, f) for f in food.asList()])

fscore = 0
if food_dist > 0:
    fscore = 10 / food_dist + 8

gscore = 0
if nearestdist > 0:
    if(sum(scaredtimes) > 10):
        gscore = 300 / nearestdist
    elif(sum(scaredtimes) > 0):
        gscore = 150 / nearestdist
    else:
        gscore = -15 / nearestdist
```

```
        gscore = -15 / nearestdist

evaluation = score + fscore + gscore
return evaluation
# raise NotImplementedError("To be implemented")
# End your code (Part 4)
```

Part II. Results & Analysis (10%):

```
D:\user\Desktop\for_school\artificial_intelligence\HW3\HW3>python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3

Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0
Win Rate:      0/1 (0.00)
Record:        Loss
```

Why the pacman will rush to the ghost is that it can't escape from these two ghosts, so it die earlier to make the rest score higher.

```
Average Score: 132.25
Scores:        516.0, -495.0, 516.0, 516.0, 516.0, -495.0, -492.0, 516.0, -492.0, 513.0, 516.0, 516.0, 516.0, -492.0, 51
6.0, 516.0, 516.0, -492.0, 516.0, 516.0, 513.0, 516.0, 512.0, -492.0, -492.0, -494.0, 516.0, 516.0, 516.0, -496.0, 516.0
, -492.0, 516.0, 513.0, -492.0, -495.0, 516.0, -492.0, 516.0, 516.0, 516.0, 516.0, 516.0, 516.0, -492.0, -496.0, -492.0,
 516.0, 513.0, -492.0, 514.0, -492.0, -492.0, -495.0, -495.0, 516.0, -492.0, 516.0, -492.0, 516.0, 516.0, 516.0, 516.0,
516.0, -495.0, -492.0, 516.0, -492.0, 516.0, -492.0, 516.0, 516.0, 516.0, 516.0, -492.0, -495.0, 516.0, 516.0, -492.0, 5
16.0, 516.0, 516.0, -496.0, 516.0, -495.0, 516.0, 516.0, -496.0, 516.0, 516.0, 516.0, 513.0, 516.0, 514.0, -495.0, 516.0
, -492.0, -495.0, -492.0, 516.0
Win Rate:      62/100 (0.62)
Record:        Win, Loss, Win, Win, Win, Loss, Loss, Win, Loss, Win, Win, Win, Win, Loss, Win, Win, Win, Loss, Win, Win,
 Win, Win, Win, Loss, Loss, Loss, Win, Win, Win, Loss, Win, Loss, Win, Win, Loss, Loss, Win, Loss, Win, Win, Win, Win, W
in, Win, Loss, Loss, Loss, Win, Win, Loss, Win, Loss, Loss, Loss, Loss, Win, Loss, Win, Loss, Win, Win, Win, Win, Win, L
oss, Loss, Win, Loss, Win, Loss, Win, Win, Win, Win, Loss, Loss, Win, Win, Loss, Win, Win, Win, Loss, Win, Loss, Win, Wi
n, Loss, Win, Win, Win, Win, Win, Win, Loss, Win, Loss, Loss, Loss, Win
```

I test the command "python pacman.py -p MinimaxAgent -l minimaxClassic -a depth=4". The win rate is 62%, fit the question request.

```
D:\user\Desktop\for_school\artificial_intelligence\HW3\HW3>python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic

Pacman emerges victorious! Score: 1190
Average Score: 1190.0
Scores:        1190.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Run in 'python pacman.py -p AlphaBetaAgent -a depth=3 -l smallClassic'

```
D:\user\Desktop\for_school\artificial_intelligence\HW3\HW3>python pacman.py -p ExpectimaxAgent -l minimaxClassic -a dept
h=3
Pacman emerges victorious! Score: 515
Average Score: 515.0
Scores:        515.0
Win Rate:      1/1 (1.00)
Record:        Win
```

Run in 'python pacman.py -p ExpectimaxAgent -l minimaxClassic -a depth=3'

```
D:\user\Desktop\for_school\artificial_intelligence\HW3\HW3>python pacman.py -p AlphaBetaAgent -l trappedClassic -a depth
=3 -q -n 10
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss

D:\user\Desktop\for_school\artificial_intelligence\HW3\HW3>python pacman.py -p ExpectimaxAgent -l trappedClassic -a dept
h=3 -q -n 10
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Average Score: 118.4
Scores:        532.0, 532.0, -502.0, 532.0, 532.0, -502.0, 532.0, 532.0, -502.0, -502.0
Win Rate:      6/10 (0.60)
Record:        Win, Win, Loss, Win, Win, Loss, Win, Win, Loss, Loss
```

In the alpha-beta agent, the win rate is 0%, while in the expectimax agent, the win rate is 60%. The reason the alpha-beta agent loses every time is that its behavior is similar to a minimax agent—it rushes toward the nearest ghost to minimize the potential loss in score. This leads to frequent collisions with the ghosts, resulting in

losing the game.

On the other hand, the expectimax agent has a 60% win rate due to its use of average value calculations, which means that it doesn't always choose the most conservative path. Instead, it considers the probabilistic outcomes of each move. This approach can lead to scenarios where the farthest ghost moves even farther away from Pacman. In those instances, Pacman can collect all the pellets before the ghost changes direction, resulting in a win.

The critical difference lies in how these agents make decisions: the alpha-beta agent leans toward the worst-case scenario, often leading to aggressive ghost-avoiding strategies, while the expectimax agent's approach allows for more nuanced outcomes, sometimes favoring riskier paths that can lead to successful pellet collection.

```
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
***     1271.0 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 ###


Finished at 20:26:52

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

I test all by autograder.py at the same time, and all of them pass the test.