Homework 3: Multi-Agent Search

109550164 徐聖哲

```
def minimax(depth, index, gameState):
    best_action = None
    best_value = None

    if index == gameState.getNumAgents():
        index = 0
        depth = depth + 1

    if depth == self.depth or gameState.isWin() or gameState.isLose():
        return None, self.evaluationFunction(gameState)

    legalMoves = gameState.getLegalActions(index)
    if len(legalMoves) == 0:
        return None, self.evaluationFunction(gameState)

    if index == 0:
        for action in legalMoves:
            next_gameState = gameState.getNextState(index, action)
            next_index= index + 1
            cur_action, cur_score = minimax(depth, next_index, next_gameState)

            if best_value is None or best_value < cur_score:
                best_value = cur_score
                best_action = action

    else:
        for action in legalMoves:
                next_gameState = gameState.getNextState(index, action)
                next_index = index + 1
                cur_action, cur_score = minimax(depth, next_index, next_gameState)

                if best_value is None or best_value > cur_score:
                    best_value = cur_score
                    best_action = action

    return best_action, best_value

action, score = minimax(0,0, gameState);
return action;
```

```
"""
1.use recurssion to implement minmax_search
2.firt if index = number of index, go to the next depth, and index turns back to pacman
3.if depth reach the end or win_state or lose_state return score
4.get the legal_action by legalmoves, if legalmoves = 0, return score
5.if index == 0, do max search -> pacman
6.best_score < cur_score, change best_score, and increase index by 1
7.if index > 0, do min search -> ghost
8.best_score > cur_score, change best_score, and increase index by 1
9.start minimax with index = 0 and depth = 0
"""
```

```python
# Begin your code (Part 1)
def alphabet(depth, index, gameState, a, b):
    if index == gameState.getNumAgents():
        index = 0
        depth = depth + 1

    if depth == self.depth or gameState.isWin() or gameState.isLose():
        return None, self.evaluationFunction(gameState)

    best_action = None
    best_value = None
    (variable) legalMoves: Any
    legalMoves = gameState.getLegalActions(index)
    if len(legalMoves) == 0:
        return None, self.evaluationFunction(gameState)

    if index == 0:
        for action in legalMoves:
            next_gameState = gameState.getNextState(index, action)
            next_index= index + 1
            cur_action, cur_score = alphabet(depth, next_index, next_gameState, a, b)

            if best_value is None or best_value < cur_score:
                best_value = cur_score
                best_action = action

            if b < best_value:
                break;

            a = max(a, best_value)
```

```
    else:
        for action in legalMoves:
            next_gameState = gameState.getNextState(index, action)
            next_index= index + 1
            cur_action, cur_score = alphabet(depth, next_index, next_gameState, a, b)

            if best_value is None or best_value > cur_score:
                best_value = cur_score
                best_action = action

            if a > best_value:
                break
            b = min(b, best_value)

    if best_value is None:
        return None, self.evaluationFunction(gameState)
    else:
        return best_action, best_value

a = float("-inf")
b = float("inf")
action, score = alphabet(0, 0, gameState, a, b);
return action;
```

```
"""
1.use recurssion to implement alphabet_search
2.firt if index = number of index, go to the next depth, and index turns back to pacman
3.if depth reach the end or win_state or lose_state return score
4.get the legal_action by legalmoves, if legalmoves = 0, return score
5.if index == 0, do max search -> pacman
6.best_score < cur_score, change best_score, and increase index by 1
7.if index > 0, do min search -> ghost
8.best_score > cur_score, change best_score, and increase index by 1
9.start alphabet with index = 0 and depth = 0
10.use a and b as alpha and beta, initiate them as -inf and inf
11.if cur_score is larger than beta, no need oto search ther nodes, return beta
12.alpha is the max_value of alpha and cur_score,check for other optical solution
13.if cur_score is smaller than alpha, no need oto search ther nodes, return alpha
14.beta is the min_value of alpha and cur_score,check for other optical solution
"""
```

```
# Begin your code (Part 3)
def expect_value(depth, index, gameState):
    if index == gameState.getNumAgents():
        index = 0
        depth = depth + 1

    if depth == self.depth or gameState.isWin() or gameState.isLose():
        return None, self.evaluationFunction(gameState)

    best_action = None
    best_value = None

    legalMoves = gameState.getLegalActions(index)
    if len(legalMoves) == 0:
        return None, self.evaluationFunction(gameState)
    else:
        p = 1/len(legalMoves)

    if index == 0:
        for action in legalMoves:
            next_gameState = gameState.getNextState(index, action)
            next_index= index + 1
            cur_action, cur_score = expect_value(depth, next_index, next_gameState)

            if best_value is None or best_value < cur_score:
                best_value = cur_score
                best_action = action
```

```
    else:
        for action in legalMoves:
                next_gameState = gameState.getNextState(index, action)
                next_index= index + 1
                cur_action, cur_score = expect_value(depth, next_index, next_gameState)

                if best_value is None :
                    best_value = 0.0

                best_value = best_value + p*cur_score
                best_action = cur_action

    if best_value is None:
        return None, self.evaluationFunction(gameState)
    else:
        return best_action, best_value
action, score = expect_value(0,0, gameState);
return action;

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

```
"""
1.use recurssion to implement expectimax_search
2.firt if index = number of index, go to the next depth, and index turns back to pacman
3.if depth reach the end or win_state or lose_state return score
4.get the legal_action by legalmoves, if legalmoves = 0, return score
5.if index == 0, do max search -> pacman
6.best_score < cur_score, change best_score, and increase index by 1
7.if index > 0, do min search -> ghost
8.best_score > cur_score, change best_score, and increase index by 1
9.start expectimax with index = 0 and depth = 0
10.use the number of legal_Action as the partion of expectation
11.when do min_search, multiply partion to cur_score
"""
```

```
# Begin your code (Part 4)
newPos = currentGameState.getPacmanPosition()
newFood = currentGameState.getFood().asList()
# newGhost = currentGameState.getGhostPositions()
newGhost = currentGameState.getGhostStates()
newCapsule = currentGameState.getCapsules()

minFood = 1e9
for food in newFood:
    if(minFood > manhattanDistance(newPos, food)):
        minFood = manhattanDistance(newPos, food)

ghostDist = 1
closeghost = 0
scaredGhostDist = []
for each in newGhost:
    if each.scaredTimer == 0:
        ghostDist = ghostDist + manhattanDistance(newPos, each.getPosition())
        if (ghostDist <= 2):
            closeghost += 1
    elif each.scaredTimer > 0:
        scaredGhostDist = scaredGhostDist + [manhattanDistance(newPos, each.getPosition())]
        if (ghostDist <= 2):
            closeghost += 1
```

```
minScaredGhostDist = 1
if len(scaredGhostDist) > 0:
    minScaredGhostDist = min(scaredGhostDist)

food = currentGameState.getNumFood() + 1
caps = len(newCapsule) + 1

winningstate = 0
if currentGameState.isWin():
    winningstate += 66666
if currentGameState.isLose():
    winningstate -= 66666

return scoreEvaluationFunction(currentGameState) * 100000 + winningstate + (food) * 900 + \
    (ghostDist) * 20 +  1.0/(minFood) * 750 +  1.0/(caps) * 100000 - 1.0 / (minScaredGhostDist) * 20 - closeghost * 300

# raise NotImplementedError("To be implemented")
# End your code (Part 4)
```

```
"""
1.use newPos and newFood to capture pacmoon and food position
2.use newGhost and newCapsule to capture ghost and capsules pasition
3.first iterate newfood and store the minimun food distance to minFood which compares to pacman
4.use ghostDist to store the total ghost distance compares to pacman
5.if ghost distance is closed in 2, I will increase the value of closeghost
6.and I specially use scaredGhostDist to store scaredGhost distance when scaredTimer is no equal to zero
7.use winningstate's value to store the state of isWin() or isLose()
8.finaly, I add score of currentGameState, winningstate and food and ghostDist multipliers for the importance because
  I think these are property that will increase score
9.and use the reciprocal of minFood, caps and minScaredGhostDist
10. I sub closeghost with multiplpier beacause I think pacman should leave the closet ghost as far as possible
"""
```

In part 4, I think that score for currentGamestate and winningstate and caps are important, because they will empact final score largely, thus I give them large specific gravity. Meanwhile, I think pacman should avoid the closeghost, so I subtract it.Finally, for ghosts distance , food number and minimum food distance, I do use the reciprocal because I think they may be large and should not influence that much, but for caps, eating capsule is also good for score, thus I give it big multiplier.