Homework 3: Multi-Agent Search

110550085房天越

Part I. Implementation (5%):

```
# Begin your code (Part 1)
'''
I implemented this function with the algorithm given in the lecture.
I start the recurrence by setting the initial depth to 1, state to
the given gameState, and agentIndex to represent the Pacman.
When the state is Lose or Win, or when the depth is larger than
self.depth, return the current state's evaluationFunction's value.
If not, then check the agentIndex, if it's 0, then it's time for the
Pacman, choose the maximum in all the possible choices, if it's not 0,
then it's time for one of the ghosts, return the minimum in all
possible choices as the optimal choice by the ghost.
'''

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

    legal_actions=state.getLegalActions(agentIndex)
    possible_choice=[]
    for action in legal_actions:
        next_state=state.getNextState(agentIndex, action)
        if(agentIndex==state.getNumAgents()-1):
            possible_choice.append(minimax(depth+1, next_state, 0))
        else:
            possible_choice.append(minimax(depth, next_state, agentIndex+1))

    #Pacman Action
    if(agentIndex==0):
        if(depth!=1):
            bestchoice=max(possible_choice)
            return bestchoice
        else:
            bestchoice=max(possible_choice)
            for i in range(len(possible_choice)):
                if(possible_choice[i]==bestchoice):
                    return legal_actions[i]

    #Ghost Action
    else:
        bestchoice=min(possible_choice)
```

```
        return bestchoice

return minimax(1, gameState, 0)
# End your code (Part 1)
```

```
# Begin your code (Part 2)
'''
Compared to Part1, I add alpha and beta, and initialize them to -inf and
inf, respectively. When checking for Pacman, if the current value is larger
than beta, return the current value(pruning), otherwise, set alpha to the
larger one of current value and the current alpha. And for the Ghosts, if
the current value is smaller than alpha, return the current value(pruning),
otherwise, set beta to the smaller one of the current value and the current beta.
'''
def ABpruning(depth, state, agentIndex, a, b):
    if(state.isLose() or state.isWin()):
        return self.evaluationFunction(state)
    elif(depth>self.depth):
        return self.evaluationFunction(state)

    legal_actions=state.getLegalActions(agentIndex)
    possible_choice=[]
    for action in legal_actions:
        next_state=state.getNextState(agentIndex, action)
        if(agentIndex==state.getNumAgents()-1):
            x=ABpruning(depth+1, next_state, 0, a, b)
        else:
            x=ABpruning(depth, next_state, agentIndex+1, a, b)

        if(agentIndex==0):
            if(x>b):
                return x
            else:
                a=max(a, x)
        else:
            if(x<a):
                return x
            else:
                b=min(b, x)

        possible_choice.append(x)

    #Pacman Action
    if(agentIndex==0):
        if(depth!=1):
            bestchoice=max(possible_choice)
            return bestchoice
```

```
        else:
            bestchoice=max(possible_choice)
            for i in range(len(possible_choice)):
                if(possible_choice[i]==bestchoice):
                    return legal_actions[i]

    #Ghost Action
    else:
        bestchoice=min(possible_choice)
        return bestchoice

return ABpruning(1, gameState, 0, float('-Inf'), float('Inf'))
# End your code (Part 2)
```

```python
# Begin your code (Part 3)
'''
This part has almost the same concept with part 1, the only difference is
that for the Ghosts, I choose the expect value instead of the minimum value.
'''
def Expectimax(depth, state, agentIndex):
    if(state.isLose() or state.isWin()):
        return self.evaluationFunction(state)
    elif(depth>self.depth):
        return self.evaluationFunction(state)

    legal_actions=state.getLegalActions(agentIndex)
    possible_choice=[]

    for action in legal_actions:
        next_state=state.getNextState(agentIndex, action)
        if(agentIndex==state.getNumAgents()-1):
            possible_choice.append(Expectimax(depth+1, next_state, 0))
        else:
            possible_choice.append(Expectimax(depth, next_state, agentIndex+1))

    #Pacman Action
    if(agentIndex==0):
        if(depth!=1):
            bestchoice=max(possible_choice)
            return bestchoice
        else:
            bestchoice=max(possible_choice)
            for i in range(len(possible_choice)):
                if(possible_choice[i]==bestchoice):
                    return legal_actions[i]

    #Ghost Choose Expect Value
    else:
        expectchoice=float(sum(possible_choice)/len(possible_choice))
        return expectchoice

return Expectimax(1, gameState, 0)

# End your code (Part 3)
```

```
# Begin your code (Part 4)
'''
In this part, I use hope to represent the value to return, and initialize
it to the current score. Then, I give some weight to the relation to the
following parameters:
1. The minimum Manhattan distance to a ghost.
2. The remaining time for ghosts to be scared.
3. The minimum Manhattan distance to a food.
4. The number of the remaining capsules.
For each weight, I test many times to get a highest score to achieve.
And for the number of capsules, I found that no matter how the weight
changes, it would yield the same result, but I still keep it there
because I think that the tendency to get a capsule is a buff for Pacman
andmight play a role in the hidden testdata.
'''
PacmanPosition=currentGameState.getPacmanPosition()
Capsules=currentGameState.getCapsules()
NumFood=currentGameState.getNumFood()
Food=currentGameState.getFood()
GhostStates=currentGameState.getGhostStates()

hope=currentGameState.getScore()

#Relation with Ghosts
minDistance=float('Inf')
for i in GhostStates:
    d=manhattanDistance(PacmanPosition, i.getPosition())
    if(d<minDistance):
        minDistance=d
RemainingTime=0
for i in GhostStates:
    RemainingTime+=i.scaredTimer
if(minDistance!=0):
    hope+=(RemainingTime*10)/minDistance
    if(RemainingTime==0):
        hope-=4/minDistance

#Relation with Food
minFoodDistance=float('Inf')
for i in Food.asList():
    d=manhattanDistance(PacmanPosition, i)
```

```
    if(d<minFoodDistance):
        minFoodDistance=d

if(NumFood!=0):
    hope+=5/minFoodDistance

#Relation with Capsules
hope-=100*len(Capsules)

return hope
# End your code (Part 4)
```

Part II. Results & Analysis (5%):

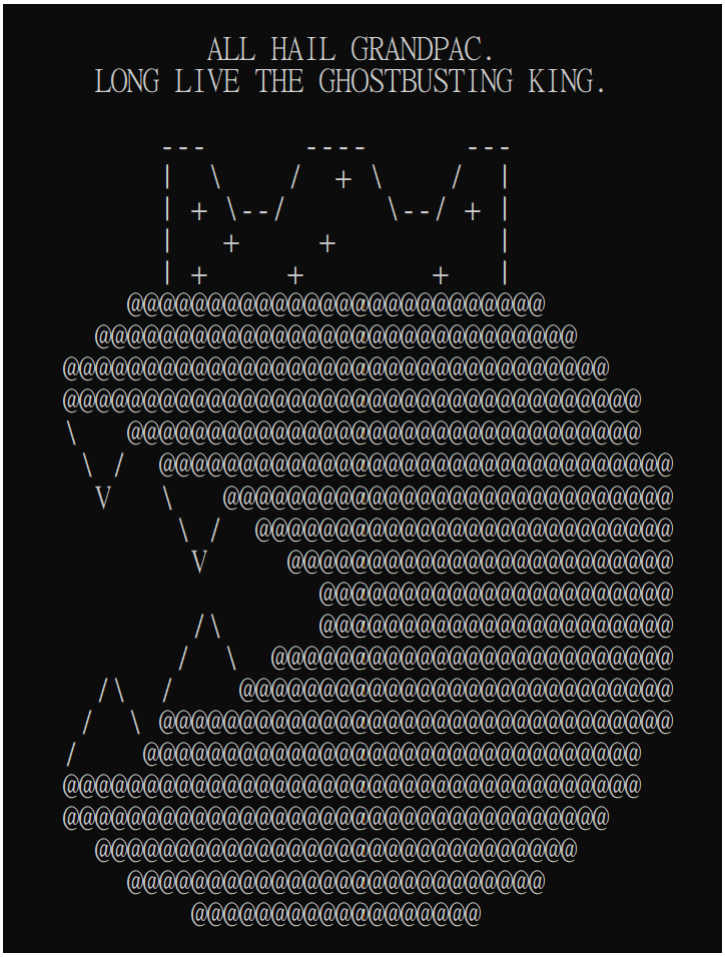
```
Question part4
==============

Pacman emerges victorious! Score: 1329
Pacman emerges victorious! Score: 1368
Pacman emerges victorious! Score: 1306
Pacman emerges victorious! Score: 1359
Pacman emerges victorious! Score: 1280
Pacman emerges victorious! Score: 1323
Pacman emerges victorious! Score: 1351
Pacman emerges victorious! Score: 1355
Pacman emerges victorious! Score: 1262
Pacman emerges victorious! Score: 1358
Average Score: 1329.1
Scores:        1329.0, 1368.0, 1306.0, 1359.0, 1280.0, 1323.0, 1351.0, 1355.0, 1262.0, 1358.0
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
***     1329.1 average score (4 of 4 points)
***         Grading scheme:
***          < 500:  0 points
***         >= 500:  2 points
***         >= 1000:  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 1:26:03

Provisional grades
==================
Question part1: 20/20
Question part2: 25/25
Question part3: 25/25
Question part4: 10/10
------------------
Total: 80/80
```



This is the best result I can get by adjusting the weights in the betterEvaluation function, I found that it is possible that I get a huge difference in one game by just adjusting a little bit of weight, maybe up to 200, but in this case, it performs relatively well in all games, and has the highest average score, so I chose that.