Homework 3: Multi-Agent Search

109550134 梁詠晴

Part I. Implementation (5%):

Part 1 :

```
    # Begin your code (Part 1)
    def minimax(depth,state,index):
        if(state.isLose() or state.isWin() or depth>self.depth): #stop and evalute when exceeding depth/ lose/ win
            return self.evaluationFunction(state)# evalute state

        legal_act = state.getLegalActions(index)# all possible action of current state & current ghost/pacman
        all_choice = []# save possible choice

        for act in legal_act : #each action
            next_state = state.getNextState(index,act)
            if((index+1) >= state.getNumAgents()):
                all_choice.append(minimax(depth+1,next_state,0)) #last ghost at current depth : repeat,append depth, index->pacman
            else:
                all_choice.append(minimax(depth,next_state,index+1)) #else : repeat, next agent

        if(index==0):#pacman
            if(depth!=1):# not top
                choice = max(all_choice)#pacman choose max
            else:
                best = max(all_choice)#pacman choose max
                for c in range(len(all_choice)):
                    if(all_choice[c]==best):
                        return legal_act[c]#return responding action of best
        elif(index > 0):# ghost
            choice = min(all_choice) # ghost choose min

        return choice # when not top return  choice after minimax
    return minimax(1,gameState,0) # start recursive
    # End your code (Part 1)
```

Part 2 :

```
    # Begin your code (Part 2)
    #raise NotImplementedError("To be implemented")
    def AlphaBeta(depth,state,index,a,b):
        # a = greatest lower bound
        # b = smallest upper bount

        if(state.isLose() or state.isWin() or depth>self.depth): #stop and evalute when exceeding depth/ lose/ win
            return self.evaluationFunction(state) # evalute state

        legal_act = state.getLegalActions(index)# all possible action of current state & current ghost/pacman
        all_choice = []# save possible choice

        for act in legal_act :  #each action
            next_state = state.getNextState(index,act)
            if((index+1) >= state.getNumAgents()):
                v=AlphaBeta(depth+1,next_state,0,a,b)  #last ghost at current depth : repeat,append depth, index->pacman
            else:
                v=AlphaBeta(depth,next_state,index+1,a,b) #else : repeat, next agent

            #pruning
            if(index==0):#pacman
                if(v>b): # if v > smallest upper bound
                    return v
                a = max(a,v)  # update new greatest lower bound if (a>v)
            if(index>0):#ghost
                if(v<a): # if v < greatest lower bound
                    return v
                b = min(b,v) # update new smallest upper bound if (b>v)
            all_choice.append(v)# update choice of remaining v
```

```
    if(index==0):#pacman
        if(depth!=1): # not top
            choice = max(all_choice)#pacman choose max
        else:
            best = max(all_choice)#pacman choose max
            for c in range(len(all_choice)):
                if(all_choice[c]==best):
                    return legal_act[c]#return responding action of best
    elif(index > 0):# ghost
        choice = min(all_choice)# ghost choose min
    return choice # when not top return choice

return AlphaBeta(1,gameState,0,float('-Inf'),float('Inf')) # start recursive with initial a = -infinity, b = infinity
# End your code (Part 2)
```

Part 3 :

```
# Begin your code (Part 3)
#raise NotImplementedError("To be implemented")
def expectimax(depth,state,index):
    if(state.isLose() or state.isWin() or depth>self.depth):#stop and evalute when exceeding depth/ lose/ win
        return self.evaluationFunction(state)  # evalute state

    legal_act = state.getLegalActions(index)# all possible action of current state & current ghost/pacman
    all_choice = [] # save possible choice

    for act in legal_act :  #each action
        next_state = state.getNextState(index,act)
        if((index+1) >= state.getNumAgents()):
            all_choice.append(expectimax(depth+1,next_state,0)) #last ghost at current depth : repeat,append depth, index->pacman
        else:
            all_choice.append(expectimax(depth,next_state,index+1)) #else : repeat, next agent


    if(index==0): #pacman
        if(depth!=1):# not top
            choice = max(all_choice)#pacman choose max
        else:
            best = max(all_choice)#pacman choose max
            for c in range(len(all_choice)):
                if(all_choice[c]==best):
                    return legal_act[c] #return responding action of best
    elif(index > 0): # ghost -> expect value
        choice = float(sum(all_choice)/len(all_choice)) # choose the choice with average value

    return choice #when not top return choice
return expectimax(1,gameState,0)
# End your code (Part 3)
```

Part 4 :

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

Pos = currentGameState.getPacmanPosition()
Food = currentGameState.getFood()
GhostStates = currentGameState.getGhostStates()
ScaredTimes = [ghostState.scaredTimer for ghostState in GhostStates]

minGhostDistance = min([manhattanDistance(Pos, state.getPosition()) for state in GhostStates])

score = currentGameState.getScore()

if(len(Food.asList())>0): #if food != 0
    NearestFoodDistance = min([manhattanDistance(Pos, food) for food in Food.asList()]) #nearest food
else:
    NearestFoodDistance = 0

if NearestFoodDistance>0 :  #if food != 0
        f_score = 10/NearestFoodDistance+5 # compute food score : higher if food is closer
else :
    f_score = 0
if minGhostDistance>0 :
    if(sum(ScaredTimes)>3): # scaredtime>3 : higher possitive score to eat ghost
        g_score = 290/minGhostDistance
    elif(sum(ScaredTimes)>0): # scaredtime almost end : slow down to avoid killed by ghost, possitive score to eat ghost
        g_score = 150/minGhostDistance
    else :
        g_score = -13/minGhostDistance # not scaredtime : minus more when the ghost is closer
else:
    g_score = 0
better = score + f_score + g_score # sum the base score, food score, ghost score

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

Part II. Results & Analysis (5%):

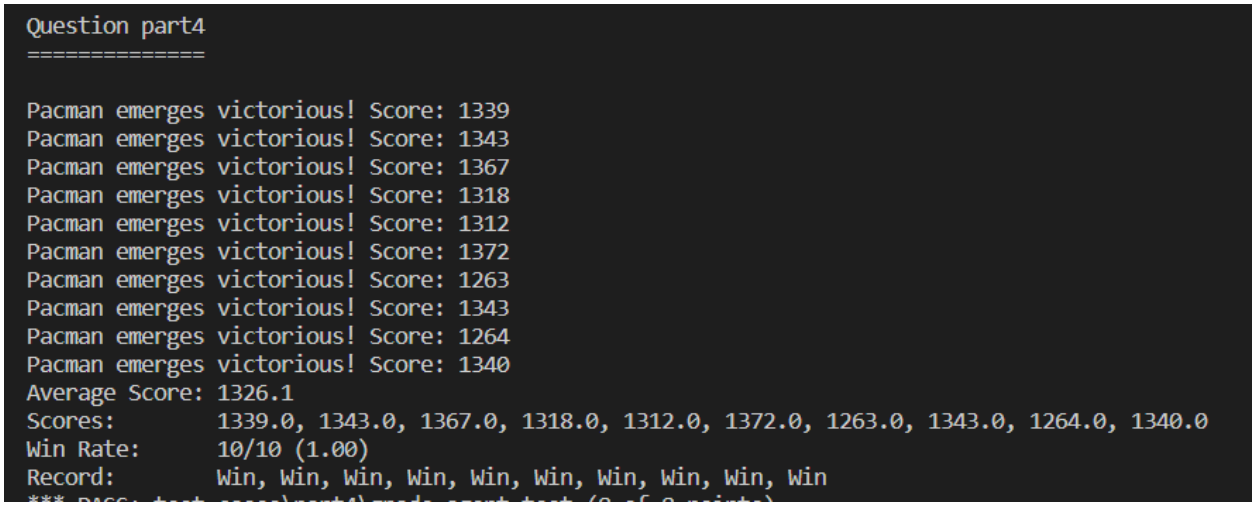
Result :

```
*** EXTRA CREDIT: 2 points
***     1326.1 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 4:28:59

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



For part 4, I design better with distance of food, distance of the ghost, and if it's scared time or not. Because eating a ghost brings much more points, I set a high value that increases the score to let pacman temporarily ignore food and chase the ghost first. However if the scared time ended and pacman was close to the ghost but hadn't eaten it , pacman sometimes will be eaten by the ghost when scared time ends, thus when the scared time is almost end, I slightly decrease the score so that this won't happen.