Homework 3: Multi-Agent Search

Part I. Implementation (5%):

```
# Begin your code (Part 1)
#raise NotImplementedError("To be implemented")
"""
Starting from depth=1 and agentIndex=0, we have to check the ghost. Go into "minimax" with agentIndex=1.
Do the same things, go into "minimax" with agentIndex=2. After checking all the pacman and ghosts in this depth
, we go to "minimax" with depth=2 and agentIndex again. Repeat the process until we meet the depth we want to search
, and the code will return evaluationFunction(gameState), the value. Because of the recursive, we trace back to agentIndex=2
and depth = self.depth-1. After returning the minimax, we returns to agentIndex=1 and do the same thing.
Only if agentIndex=0, we returns the maximum. If depth=1 and agentIndex=0, it means we have finished the function
and we have to return the action that we should do.
"""
```

```
def minimax(depth, agentIndex, gameState):
    if (gameState.isWin() or gameState.isLose() or depth > self.depth):
        return self.evaluationFunction(gameState)
    ret = [] # store the value of the state after the actions
    todo = gameState.getLegalActions(agentIndex) # store all the legal actions in this gamestate and agentIndex
    for action in todo:
        successor = gameState.getNextState(agentIndex, action) #after doing the action, successor stores next state
        if((agentIndex+1) >= gameState.getNumAgents()):
            ret = ret + [minimax(depth+1, 0, successor)]
        else:
            ret = ret + [minimax(depth, agentIndex+1, successor)]
    if agentIndex == 0:
        if(depth == 1):
            maxscore = max(ret)
            for i in range(len(ret)):
                if (ret[i] == maxscore):
                    return todo[i]
        else:
            retVal = max(ret)
    elif agentIndex > 0:
        retVal = min(ret)
    return retVal
return minimax(1,0,gameState)
# End your code (Part 1)
```

```
# Begin your code (Part 2)
#raise NotImplementedError("To be implemented")
"""
AlphaBeta is almost the same as minimax. The only difference is that we have alpha and beta.
Beta means best option from min node to the root.
Alpha means best option from max node to the root.
"""
def alphaBeta(depth, agentIndex, gameState, a, b):
    alpha = a
    beta = b
    if (gameState.is (variable) evaluationFunction: Any  > self.depth):
        return self.evaluationFunction(gameState)
    retList = []
    todo = gameState.getLegalActions(agentIndex)
    for action in todo:
        successor = gameState.getNextState(agentIndex, action)
        if((agentIndex+1) >= gameState.getNumAgents()):
            ret = alphaBeta(depth+1, 0, successor, alpha, beta)
        else:
            ret = alphaBeta(depth, agentIndex+1, successor, alpha, beta)
```

```
            if(agentIndex == 0 and ret > beta): # cut the node
                return ret
            if (agentIndex > 0 and ret < alpha): # cut the node
                return ret
            if (agentIndex == 0 and ret > alpha): # replace
                alpha = ret
            if (agentIndex > 0 and ret < beta): # replace
                beta = ret
            retList += [ret]
        if agentIndex == 0:
            if(depth == 1):
                maxscore = max(retList)
                length = len(retList)
                for i in range(length):
                    if (retList[i] == maxscore):
                        return todo[i]
            else:
                retVal = max(retList)

        elif agentIndex > 0:
            retVal = min(retList)
        return retVal
    return alphaBeta(1,0, gameState, -99999, 99999)
    # End your code (Part 2)
```

```
    # Begin your code (Part 3)
    #raise NotImplementedError("To be implemented")
    """
    It's also the same as minimax, but the ghosts will move ramdomly.
    As a result, the value of the ghosts is the average of values of all the actions.
    """
```

```
def performExpectimax(depth, agentIndex, gameState):
    if (gameState.isWin() or gameState.isLose() or depth > self.depth):
        return self.evaluationFunction(gameState)
    ret = []
    todo = gameState.getLegalActions(agentIndex)
    for action in todo:
        successor = gameState.getNextState(agentIndex, action)
        if((agentIndex+1) >= gameState.getNumAgents()):
            ret += [performExpectimax(depth+1, 0, successor)]
        else:
            ret += [performExpectimax(depth, agentIndex+1, successor)]
    if agentIndex == 0:
        if(depth == 1):
            maxscore = max(ret)
            length = len(ret)
            for i in range(length):
                if (ret[i] == maxscore):
                    return todo[i]
        else:
            retVal = max(ret)
    elif agentIndex > 0:
        s = sum(ret)
        l = len(ret)
        retVal = float(s/l)
    return retVal
return performExpectimax(1, 0, gameState)
# End your code (Part 3)
```

```
# Begin your code (Part 4)
"""
The nearest the food, the highest the score. <- score += food / min(distancesToFoodList)
The nearest the ghosts, the lower the schore (because ghost_weight is negative) <- score += ghost_weight / distance
However, if the ghost is scared, the score will be very high. <- score += scared_ghost_weight / distance
"""
```

```
Pos = currentGameState.getPacmanPosition()
Food = currentGameState.getFood()
GhostStates = currentGameState.getGhostStates()
INF = 100000000.0
food_weight = 10.0
ghost_weight = -10.0
scared_ghost_weight = 200.0
score = currentGameState.getScore()
distancesToFoodList=[]
for foodPos in Food.asList():
    distancesToFoodList = [util.manhattanDistance(Pos, foodPos) ] # distance from all the food to pacman
if len(distancesToFoodList) > 0:
    score = score + food_weight / min(distancesToFoodList)
else:
    score = score + food_weight
for ghost in GhostStates:
    distance = manhattanDistance(Pos, ghost.getPosition()) # distance from all the ghosts to pacman
    if distance > 0:
        if ghost.scaredTimer > 0:
            score = score + scared_ghost_weight / distance
        else:
            score = score + ghost_weight / distance
    else:
        return -INF
return score
# End your code (Part 4)
```

Part II. Results & Analysis (5%):