Homework 3: Multi-Agent Search 109550171 陳存佩

Part I. Implementation (5%):

Part 1

```python
class MinimaxAgent(MultiAgentSearchAgent):
    """Your minimax agent (Part 1)"""
    def getAction(self, gameState):
        """...
        # Begin your code (Part 1)
        def performMinimax(depth, agentIndex, gameState):
            if (gameState.isWin() or gameState.isLose() or depth > self.depth): # Terminal condition (win or lose or exceed depth)
                return self.evaluationFunction(gameState) # Get evaluation

            value = []  # Store the values for this node
            todo = gameState.getLegalActions(agentIndex)  # Get all legal actions (a list) of an agent
            for action in todo:
                successor = gameState.getNextState(agentIndex, action) # The gameState after taking the legal action
                if((agentIndex+1) >= gameState.getNumAgents()): # When all agents are done
                    value += [performMinimax(depth+1, 0, successor)] # Pacman go to the next level
                else:
                    value += [performMinimax(depth, agentIndex+1, successor)]
                    # Calculate ghosts's min value for this successor(next possible state)
                    # Ghost n will use ghost n-1's successors. After getting the last ghost's value, we go backtracking.

            if agentIndex == 0: # Pacman
                if(depth == 1): # Back to root : return action
                    for i in range( len(value) ):
                        if (value[i] == max(value)):
                            return todo[i]
                else:
                    return max(value) # Not a root : return max value
            elif agentIndex > 0: # Ghosts : return min value
                return min(value)

        return performMinimax(1, 0, gameState) # Go to the function with agent 0 (pacman),depth 1
        # End your code (Part 1)
```

Part 2

```python
class AlphaBetaAgent(MultiAgentSearchAgent):
    """Your minimax agent with alpha-beta pruning (Part 2)"""
    def getAction(self, gameState):
        """Returns the minimax action using self.depth and self.evaluationFunction"""
        # Begin your code (Part 2)
        def performAlphaBeta(depth, agentIndex, gameState, alpha, beta): # alpha : best for max, beta : best for min
            if (gameState.isWin() or gameState.isLose() or depth > self.depth): # Terminal condition (win or lose or exceed depth)
                return self.evaluationFunction(gameState) # Get evaluation

            valueList = []  # Store the values for this node
            todo = gameState.getLegalActions(agentIndex) # Get all legal actions (a list) of an agent
            for action in todo:
                successor = gameState.getNextState(agentIndex, action) # The gameState after taking the legal action
                if((agentIndex+1) >= gameState.getNumAgents()): # When all agents are done
                    value = performAlphaBeta(depth+1, 0, successor, alpha, beta) # Pacman go to the next level
                else:
                    value = performAlphaBeta(depth, agentIndex+1, successor, alpha, beta)
                    # Calculate ghosts's min value for this successor(next possible state) (We carry the current alpha and beta)
                    # Ghost n will use ghost n-1's successors. After getting the last ghost's value, we go backtracking.
                if(agentIndex == 0 and value > beta): # Pacman : impossible to go this branch : cut
                    return value
                if (agentIndex > 0 and value < alpha): # Ghost : impossible to go this branch : cut
                    return value
                if (agentIndex == 0 and value > alpha): # Pacman : find value > alpha : replace it
                    alpha = value
                if (agentIndex > 0 and value < beta): # Ghost : find value < beta : replace it
                    beta = value
                valueList += [value]

            if agentIndex == 0: # Pacman
                if(depth == 1): # Back to root : return action
                    for i in range(len(valueList)):
                        if (valueList[i] == max(valueList)):
                            return todo[i]
                else:
                    return max(valueList) # Not a root : return max value
            elif agentIndex > 0: # Ghosts
                return min(valueList) # Ghosts : return min value
        return performAlphaBeta(1, 0, gameState, -99999, 99999) # Go to the function with very small alpha, very big beta
        # End your code (Part 2)
```

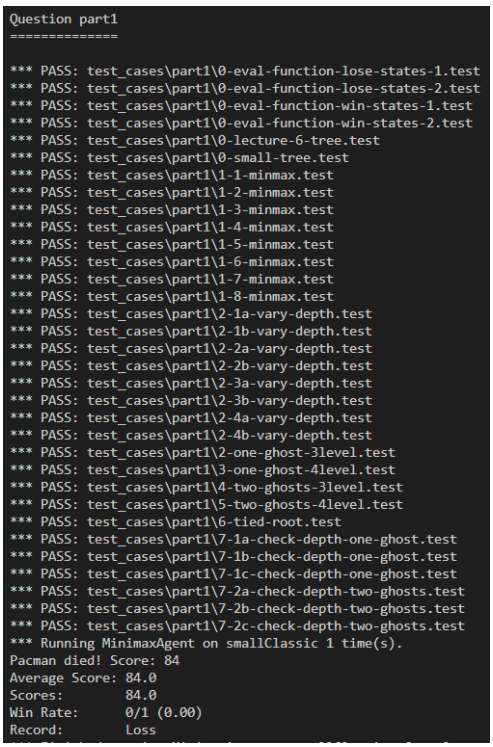
Part 3

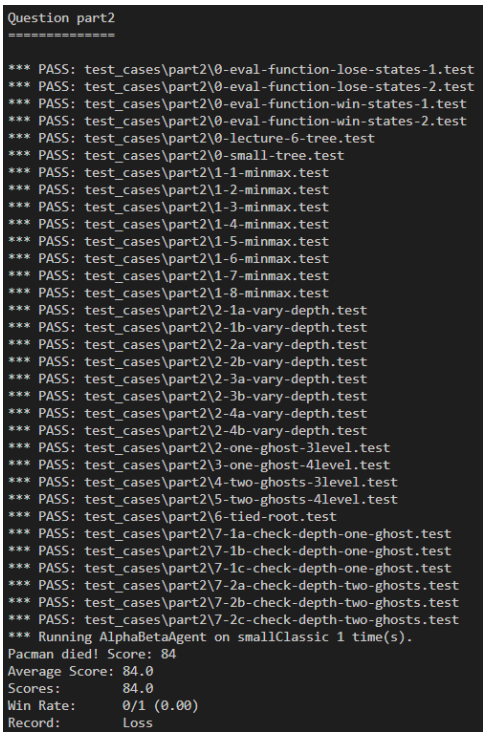
```python
class ExpectimaxAgent(MultiAgentSearchAgent):
    """Your expectimax agent (Part 3)"""
    def getAction(self, gameState):
        """
        Returns the expectimax action using self.depth and self.evaluationFunction
        All ghosts should be modeled as choosing uniformly at random from their  legal moves.
        """
        # Begin your code (Part 3)
        def performExpectimax(depth, agentIndex, gameState):
            if (gameState.isWin() or gameState.isLose() or depth > self.depth): # Terminal condition (win or lose or exceed depth)
                return self.evaluationFunction(gameState) # Get evaluation

            value = []  # Store the values for this node
            todo = gameState.getLegalActions(agentIndex) # Get all legal actions (a list) of an agent

            for action in todo:
                successor = gameState.getNextState(agentIndex, action) # The gameState after taking the legal action
                if((agentIndex+1) >= gameState.getNumAgents()): # When all agents are done
                    value += [performExpectimax(depth+1, 0, successor)] # Pacman go to the next level
                else:
                    value += [performExpectimax(depth, agentIndex+1, successor)]
                    # Calculate ghosts's expected utilities for this successor(next possible state)
                    # Ghost n will use ghost n-1's successors. After getting the last ghost's expected utilities, we go backtracking.
            if agentIndex == 0: # Pacman
                if(depth == 1): # Back to root : return action
                    for i in range(len(value)):
                        if (value[i] == max(value)):
                            return todo[i]
                else:
                    return max(value) # Not a root : return max value
            elif agentIndex > 0: # Ghosts
                s = sum(value)
                l = len(value)
                return float(s/l) # Ghosts : return expected utilities
        return performExpectimax(1, 0, gameState) # Go to the function with agent 0 (pacman), depth 1
        # End your code (Part 3)
```

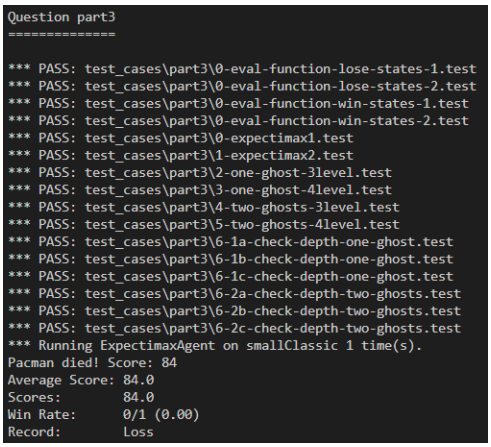
Part 4

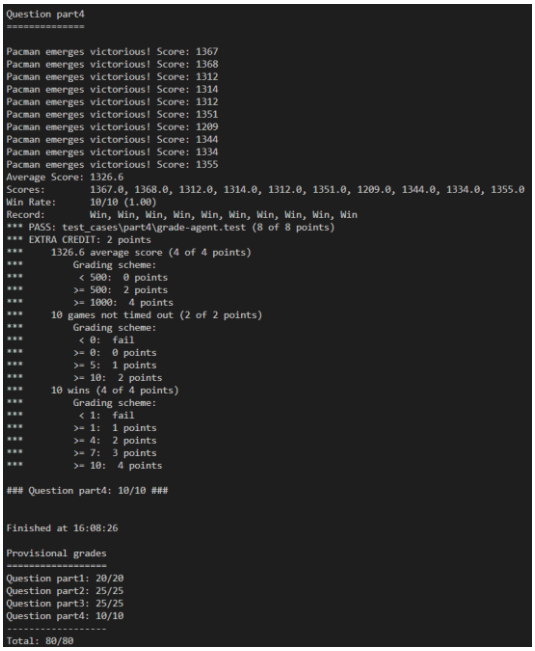
```python
def betterEvaluationFunction(currentGameState):
    """Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable evaluation function (Part 4)."""
    # Begin your code (Part 4)
    newPos = currentGameState.getPacmanPosition() # Get pacman position data
    newFood = currentGameState.getFood() # Get food position data
    newCapsules = currentGameState.getCapsules() # Get capsule position data
    newGhostStates = currentGameState.getGhostStates() # Get ghosts position data
    #Set up the weight
    INF = 1000000000000.0
    WEIGHT_FOOD = 5.0
    WEIGHT_CAP = 10.0
    WEIGHT_GHOST = -10.0
    WEIGHT_SCARED_GHOST = 200.0
    adjust_score = currentGameState.getScore() # Adjust score is modified from base score

    distancesToFoodList = [util.manhattanDistance(newPos, foodPos) for foodPos in newFood.asList()] # Calculate the distances to food
    if len(distancesToFoodList) > 0:
        adjust_score += ( WEIGHT_FOOD / min(distancesToFoodList) ) # Consider the closest food. The food is closer, the score is higher.

    distancesToCapList = []
    for ii in range(len(newCapsules)):
        distancesToCapList.append(abs(newPos[0] - newCapsules[ii][0]) + abs(newPos[1] - newCapsules[ii][1])) #Calculate the distances between pacman and capsule
        if len(distancesToCapList) > 0:
            adjust_score += WEIGHT_CAP / min(distancesToCapList) # Consider the closest capsule

    for ghost in newGhostStates:
        distance = manhattanDistance(newPos, ghost.getPosition()) # Calculate the distances to ghost
        if distance > 0:
            if ghost.scaredTimer > 1:
                adjust_score += WEIGHT_SCARED_GHOST / distance # In scared time : the ghost is closer, the score is higher.
            elif ghost.scaredTimer > 0:
                adjust_score += (WEIGHT_SCARED_GHOST - 50) / distance # Scared time is about to end : WEIGHT decreases
            else:
                adjust_score += WEIGHT_GHOST / distance # Not in scared time : the ghost is closer, the score is lower. (WEIGHT_GHOST is negative)
        else: # Distance <= 0 : pacman died
            return -INF
    return adjust_score
    # End your code (Part 4)
```

Part II. Results & Analysis (5%):









My evaluation function :

I consider the following 4 elements :

- Food : I consider the closest food. Eating the closest food first can save time, it's like a kind of greedy algorithm idea.
- Capsule : I also consider the capsules. Capsules have special effect, so its weight is higher than food.
- Ghost : Pacman have to avoid encountering ghosts when it's not scared time, so the WEIGHT_GHOST is negative.
- Scared time : In scared time, pacman eating ghosts can get much more score than eating food, so I assign higher weight for WEIGHT_SCARED_GHOST than WEIGHT_FOOD.
  When the scared time is about to end, I decrease the WEIGHT_SCARED_GHOST to avoid encountering the ghost when time's up.