Homework 3: Multi-Agent Search

Part I. Implementation (5%):

- **Part 1 MiniMax**

```python
class MinimaxAgent(MultiAgentSearchAgent):
    """
    Your minimax agent (Part 1)
    """

    def getAction(self, gameState):
        """
        Returns the minimax action from the current gameState using self.depth
        and self.evaluationFunction.
        """
        # Begin your code (Part 1)
        # raise NotImplementedError("To be implemented")
        __, nextAction = self.performMinimax(0, self.index, gameState)
        return nextAction

    def performMinimax(self, depth, agentIndex, gameState):
        # Return the score if the game is over or the depth is reached
        if (gameState.isWin() or gameState.isLose() or depth >= self.depth):
            return self.evaluationFunction(gameState), None

        # Init variables and lists
        scores = []
        isMinplayer = True if agentIndex != 0 else False
        bestScore = float("inf") if isMinplayer else float("-inf")
        legalMoves = gameState.getLegalActions(agentIndex)
        nextIndex = (agentIndex + 1) % gameState.getNumAgents() # 0 -> 1 -> 2 -> 0 -> 1 -> 2 -> ...

        # Increase depth if the next agent is pacman
        if nextIndex == 0:
            depth +=1

        # Remove the stop action to avoid some extreme cases
        if Directions.STOP in legalMoves:
            legalMoves.remove(Directions.STOP)

        # Perform minimax
        for move in legalMoves:
            nextState = gameState.getNextState(agentIndex, move)
            nextValue, __ = self.performMinimax(depth, nextIndex, nextState)
            bestScore = min(bestScore, nextValue) if isMinplayer else max(bestScore, nextValue)
            scores.append(nextValue)

        # Return the best score and the next action
        if isMinplayer:
            return bestScore, None
        else:
            bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
            choisenIndex = random.choice(bestIndices)
            return max(scores), legalMoves[choisenIndex]
        # End your code (Part 1)
```

- **Part 2 Alpha-Beta**

```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)
        # raise NotImplementedError("To be implemented")
        alpha, beta = float("-inf"), float("inf")
        __, nextAction = self.performAlphaBeta(0, self.index, gameState, alpha, beta)
        return nextAction

    def performAlphaBeta(self, depth, agentIndex, gameState, alpha, beta):
        # Return the score if the game is over or the depth is reached
        if (gameState.isWin() or gameState.isLose() or depth >= self.depth):
            return self.evaluationFunction(gameState), None

        # Init variables and lists
        scores = []
        isMinplayer = True if agentIndex != 0 else False
        bestScore = float("inf") if isMinplayer else float("-inf")
        legalMoves = gameState.getLegalActions(agentIndex)
        nextIndex = (agentIndex + 1) % gameState.getNumAgents() # 0 -> 1 -> 2 -> 0 -> 1 -> 2 -> ...

        # Increase depth if the next agent is pacman
        if nextIndex == 0:
            depth +=1

        # Remove the stop action to avoid some extreme cases
        if Directions.STOP in legalMoves:
            legalMoves.remove(Directions.STOP)

        # Perform alpha-beta pruning
        for move in legalMoves:
            nextState = gameState.getNextState(agentIndex, move)
            nextValue, __ = self.performAlphaBeta(depth, nextIndex, nextState, alpha, beta)
            bestScore = min(bestScore, nextValue) if isMinplayer else max(bestScore, nextValue)

            # Pruning
            if isMinplayer :
                if nextValue < alpha:
                    return nextValue, None
                beta = min(beta, bestScore)
            else:
                if nextValue > beta:
                    return nextValue, move
                alpha = max(alpha, nextValue)
            scores.append(nextValue)

        # Return the best score and the next action
        if isMinplayer:
            return bestScore, None
        else:
            bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
            choisenIndex = random.choice(bestIndices)
            return max(scores), legalMoves[choisenIndex]
        # End your code (Part 2)
```

- **Part3 Expectimax**

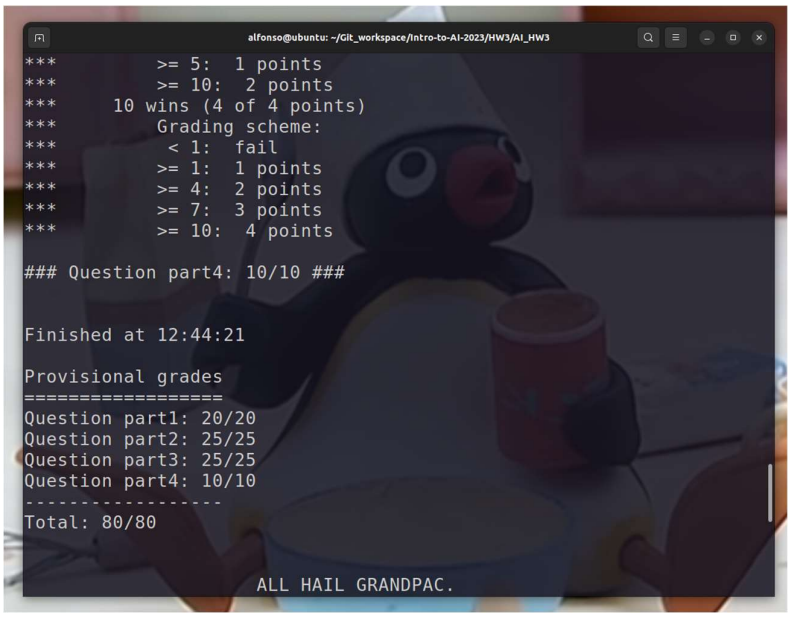
```python
class ExpectimaxAgent(MultiAgentSearchAgent):
    """
      Your expectimax agent (Part 3)
    """

    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)
        # raise NotImplementedError("To be implemented")
        __, nextAction = self.performExpectimax(0, self.index, gameState)
        return nextAction

    def performExpectimax(self, depth, agentIndex, gameState):
        # Return the score if the game is over or the depth is reached
        if (gameState.isWin() or gameState.isLose() or depth >= self.depth):
            return self.evaluationFunction(gameState), None

        # Init variables and lists
        scores = []
        isMinplayer = True if agentIndex != 0 else False
        bestScore = float("inf") if isMinplayer else float("-inf")
        legalMoves = gameState.getLegalActions(agentIndex)
        nextIndex = (agentIndex + 1) % gameState.getNumAgents() # 0 -> 1 -> 2 -> 0 -> 1 -> 2 -> ...

        # Increase depth if the next agent is pacman
        if nextIndex == 0:
            depth +=1

        # Remove the stop action to avoid some extreme cases
        if Directions.STOP in legalMoves:
            legalMoves.remove(Directions.STOP)

        # Perform expectimax
        for move in legalMoves:
            nextState = gameState.getNextState(agentIndex, move)
            nextValue, __ = self.performExpectimax(depth, nextIndex, nextState)
            bestScore = min(bestScore, nextValue) if isMinplayer else max(bestScore, nextValue)
            scores.append(nextValue)

        # Return the best score and the next action
        if isMinplayer:
            s = sum(scores)
            l = len(scores)
            return s/l, None
        else:
            bestIndices = [index for index in range(len(scores)) if scores[index] == bestScore]
            choisenIndex = random.choice(bestIndices)
            return max(scores), legalMoves[choisenIndex]
        # End your code (Part 3)
```

- **Part4 Better evaluation function**

```python
def betterEvaluationFunction(currentGameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (Part 4).
    """
    # Begin your code (Part 4)
    # raise NotImplementedError("To be implemented")
    # Score
    curScore = currentGameState.getScore()

    # Ghosts
    pacmanPos = currentGameState.getPacmanPosition()
    ghostsState = [(manhattanDistance(pacmanPos, currentGameState.getGhostPosition(Id)), Id)\
                   for Id in range(1, currentGameState.getNumAgents())]
    minGhostDist, minGhostId = (0, 0) if len(ghostsState) == 0 else min(ghostsState)
    isScared = currentGameState.data.agentStates[minGhostId].scaredTimer > 1

    # Food
    foodDist = [manhattanDistance(pacmanPos, food) for food in currentGameState.getFood().asList()]
    numFood = currentGameState.getNumFood()
    minFoodDist = 1 if numFood == 0 else min(foodDist)

    # Capsules
    numCapsules = len(currentGameState.getCapsules())
    capsulesDist = [manhattanDistance(pacmanPos, capsule) for capsule in currentGameState.getCapsules()]
    minCapsuleDist = float(9999999) if len(currentGameState.getCapsules()) == 0 else min( capsulesDist )

    # Init variables and weight
    variables = [curScore, minGhostDist, numCapsules, minCapsuleDist, minFoodDist, numFood]
    weight = [  [11, -6, 0, 0, 0, 0],    # isScared
                [9, 0, -5, -5, -1, -1], # minCapsuleDist < 6
                [10, 0, 0, 0, -4, -4],  # numCapsules == 0
                [9, 1, -5, -5, -4, -4]  # default
            ]

    # Return the score
    if isScared:
        weightNum = 0
    elif minCapsuleDist < 6 :
        weightNum = 1
    elif numCapsules == 0:
        weightNum = 2
    else :
        weightNum = 3 # default
    return sum([ x*y for x, y in zip(variables, weight[weightNum]) ])
    # End your code (Part 4)
```

Part II. Results & Analysis (5%):

- **The screen shot of the result.**



- **Analysis**
- **The main goal of this evaluation function is to achieve the highest possible score. Therefore, Pac-Man is designed to chase the target with the highest score, which includes scared ghosts, capsules, and food in a priority sequence from strong to weak. By applying a negative parameter to the number of prey and adding it to the evaluation function's return score, Pac-Man will be more likely to chase the target. The reason for this is that if the number of preys decrease after Pac-Man's move, the negative parameter will result in a higher return score, making the move more likely to be chosen.**
- **If there are any scared ghosts, Pac-Man will desperately chase them while avoiding unscared ghosts. In the absence of scared ghosts, Pac-Man will continue eating dots, but if a capsule appears within six units of distance, Pac-Man will switch to eating the capsule. If there are no capsules left, Pac-Man will focus on eating dots.**

- In a previous version, Pac-Man had a main problem of repeatedly turning right and left near the capsule, waiting for the ghost to come close enough to eat the capsule. This problem led to a lower score due to the time wasted waiting for the ghost. After debugging the if-else condition, I realized that the evaluation function evaluated the move's result, and the condition used had not yet occurred.
- The main reason for this problem was that the evaluation function told Pac-Man that the ghost was scared, which would happen after Pac-Man made the move. This caused Pac-Man to switch targets to the ghost. After the Pac-Man left for a small distance, the evaluation function returned a value indicating that there were capsules within six units of distance, causing Pac-Man to switch back to the capsule again. I fixed the bug by increasing the score under the condition of any capsule being closer than six units of distance. This make Pac-Man choose to eat the capsule rather than chase for the ghost before the capsule is eaten.

- The Pac-Man turn repeatedly.