Homework 3: Multi-Agent Search

Part I. Implementation (5%):

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.

        Here are some method calls that might be useful when implementing minimax.

        gameState.getLegalActions(agentIndex):
        Returns a list of legal actions for an agent
        agentIndex=0 means Pacman, ghosts are >= 1

        gameState.getNextState(agentIndex, action):
        Returns the child game state after an agent takes an action

        gameState.getNumAgents():
        Returns the total number of agents in the game

        gameState.isWin():
        Returns whether or not the game state is a winning state

        gameState.isLose():
        Returns whether or not the game state is a losing state
        """
        # Begin your code (Part 1)
        def minimax(gameState, agent, depth):
            if gameState.isWin() or gameState.isLose() or depth == self.depth:
                return self.evaluationFunction(gameState), Directions.STOP
            '''
            Check if the game is over.
            '''
            if agent == 0: # pacman (max)
                return max_value(gameState, agent, depth)
            else: # ghosts (min)
                return min_value(gameState, agent, depth)
```

```
            '''
            Pacman's index is always 0, which means that we need to evaluate maximum value. On the
            contarary, indexes that are larger than 0 represent ghosts, so we need to evalute minimum
            value instead.
            '''
        '''
        General:
        initialize v = -∞
        for each a in Actions(s):
            v = max(v, value(T(s,a)))
        return v
        '''
        def max_value(gameState, agent, depth):
            maxScore = -1e9
            bestAction = NULL
            actions = gameState.getLegalActions(agent)
            '''
            Find all possible moves
            '''
            for action in actions:
                nextState = gameState.getNextState(agent, action)
                nextAgent = agent + 1
                nextDepth = depth
                if nextAgent == gameState.getNumAgents(): # last ghost
                    nextAgent = 0
                    nextDepth += 1
                '''
                Multiagents, so we need to take turns to take action. When the successor's index equals
                to the number of agents, it means that the successor is the last ghost, and a whole
                round is finished. As a result, set the successor's index to 0, and add 1 to the depth.
                '''
                curScore = minimax(nextState, nextAgent, nextDepth)[0] # score
                if curScore > maxScore:
                    maxScore = curScore
                    bestAction = action
                '''
                Save the max value for every loop, and also update the best decision move for current
                agent
                '''

            return maxScore, bestAction
```

```
        '''
        General:
        initialize v = +∞
        for each a in Actions(s):
            v = min(v, value(T(s,a)))
        return v
        '''
        def min_value(gameState, agent, depth):
            minScore = 1e9
            bestAction = NULL
            actions = gameState.getLegalActions(agent)
            for action in actions:
                nextState = gameState.getNextState(agent, action)
                nextAgent = agent + 1
                nextDepth = depth
                if nextAgent == gameState.getNumAgents(): # last ghost
                    nextAgent = 0
                    nextDepth += 1

                curScore = minimax(nextState, nextAgent, nextDepth)[0] # score
                if curScore < minScore:
                    minScore = curScore
                    bestAction = action
                '''
                Almost the same as the max_value(), the only difference is that we save the min value
                for every loop. (and the initial value is +inf, while the initial value of max_value()
                is -inf
                '''

            return minScore, bestAction
```

```
        '''
        Return the best move dicided. As minimax() would return a tuple of [score, action], we only
        need action, which is minimax()[1] for this function's return.
        '''
        move = minimax(gameState, 0, 0)[1]
        # [score, action]
        return move

        raise NotImplementedError("To be implemented")
        # End your code (Part 1)
```

AlphaBeta:

```
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)
        '''
        Same as minimax
        '''
        def AlphaBeta(gameState, agent, depth, alpha, beta):
            if gameState.isWin() or gameState.isLose() or depth == self.depth:
                return self.evaluationFunction(gameState), Directions.STOP

            if agent == 0: # pacman (max)
                return max_value(gameState, agent, depth, alpha, beta)
            else: # ghosts (min)
                return min_value(gameState, agent, depth ,alpha, beta)

        '''
```

```
        '''
        def max-value(state, α, β):
            initialize v = -∞
            for each successor of state:
                v = max(v, value(successor, α, β))
                if v ≥ β
                    return v
                α = max(α, v)
                return v
        '''
        def max_value(gameState, agent, depth, alpha, beta):
            maxScore = -1e9
            bestAction = NULL
            actions = gameState.getLegalActions(agent)
            for action in actions:
                nextState = gameState.getNextState(agent, action)
                nextAgent = agent + 1
                nextDepth = depth
                if nextAgent == gameState.getNumAgents(): # last ghost
                    nextAgent = 0
                    nextDepth += 1

                curScore = AlphaBeta(nextState, nextAgent, nextDepth, alpha, beta)[0] # score
                if curScore > maxScore:
                    maxScore = curScore
                    bestAction = action
                alpha = max(alpha, maxScore)
                if maxScore > beta:
                    break
                '''
                Decide whether we should prune it or not. If the current value(score) is greater than(or equal to) beta, then it means
                that there is no need to keep traversing the remaining successors. Because if the conditon holds, the next minimum value
                (min node's turn) has already been determined, and no matter how large (or small) the next successor's value is, that
                value won't be chosen as the final value, which allows us to prune it.
                '''

            return maxScore, bestAction
```

```python
    '''
    def min-value(state , α, β):
        initialize v = +∞
        for each successor of state:
            v = min(v, value(successor, α, β))
            if v ≤ α
                return v
            β = min(β, v)
        return v
    '''
    def min_value(gameState, agent, depth, alpha, beta):
        minScore = 1e9
        bestAction = NULL
        actions = gameState.getLegalActions(agent)
        for action in actions:
            nextState = gameState.getNextState(agent, action)
            nextAgent = agent + 1
            nextDepth = depth
            if nextAgent == gameState.getNumAgents(): # last ghost
                nextAgent = 0
                nextDepth += 1

            curScore = AlphaBeta(nextState, nextAgent, nextDepth, alpha, beta)[0] # score
            if curScore < minScore:
                minScore = curScore
                bestAction = action
            beta = min(beta, minScore)
            if minScore < alpha:
                break
            '''
            Similar to max_value(), but this time we use alpha to check if we can prune the nodes. Since the next successor of the
            min node is max node, any value that is smaller than alpha would not be chosen as the final value, so the remaining part
            can be pruned.
            '''
        return minScore, bestAction
```

```python
        '''
        Same as minimax, but need to assign an initial value of alpha and beta
        '''
        move = AlphaBeta(gameState, 0, 0, -1e10, 1e10)[1]
        #                                 alpha  beta
        # [score, action]
        return move
        raise NotImplementedError("To be implemented")
        # End your code (Part 2)
```

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)
        '''
        Same as minimax
        '''
        def expectimax(gameState, agent, depth):
            if gameState.isWin() or gameState.isLose() or depth == self.depth:
                return self.evaluationFunction(gameState), None

            if agent == 0: # pacman
                return max_value(gameState, agent, depth)
            else: # ghosts
                return expected_value(gameState, agent, depth)
```

```python
        '''
        The max_value() of expextimax is same as minimax, since the pacman would always do the best choice
        '''
        def max_value(gameState, agent, depth):
            maxScore = -1e9
            bestAction = NULL
            actions = gameState.getLegalActions(agent)
            for action in actions:
                nextState = gameState.getNextState(agent, action)
                nextAgent = agent + 1
                nextDepth = depth
                if nextAgent == gameState.getNumAgents(): # last ghost
                    nextAgent = 0
                    nextDepth += 1

                curScore = expectimax(nextState, nextAgent, nextDepth)[0] # score
                if curScore > maxScore:
                    maxScore = curScore
                    bestAction = action

            return maxScore, bestAction
```

```python
        '''
        This part is similar to min_value() in minimax. However, since the ghosts' actions are unknown, instead of assuming that they
        would always do the worst (in terms of pacman) choice, we evaluate their expectation to predict the moves and deal with the
        result.
        '''
        def expected_value(gameState, agent, depth):
            expectedScore = 0
            expectedAction = NULL
            actions = gameState.getLegalActions(agent)
            for action in actions:
                nextState = gameState.getNextState(agent, action)
                nextAgent = agent + 1
                nextDepth = depth
                if nextAgent == gameState.getNumAgents(): # last ghost
                    nextAgent = 0
                    nextDepth += 1

                '''
                Calculate the expextation of every move. In the fuction, I assume it is uniformly distributed.
                '''
                curScore = expectimax(nextState, nextAgent, nextDepth)[0] # score
                expectedScore += curScore * (1 / len(actions))

            return expectedScore, expectedAction

        '''
        Same as minimax
        '''
        move = expectimax(gameState, 0, 0)[1]
        # [score, action]
        return move

        raise NotImplementedError("To be implemented")
        # End your code (Part 3)
```

betterEvaluationFunction:

```python
def betterEvaluationFunction(currentGameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (Part 4).
    """
    # Begin your code (Part 4)
    '''
    Get all the remaining food's distance from the pacman, and find the smallest one.
    '''
    curPos = currentGameState.getPacmanPosition()
    foods = currentGameState.getFood()
    foodList = foods.asList()
    foodNum = len(foodList)
    minDist = 1e9
    for foodPos in foodList:
        dist = util.manhattanDistance(curPos, foodPos)
        if minDist > dist:
            minDist = dist

    '''
    Get the ghost's distance from the pacman, and if the distance is too close, escapse.
    '''
    ghostPos = currentGameState.getGhostPositions()
    ghostDist = 0
    for ghost in ghostPos:
        dist = util.manhattanDistance(curPos, ghost)
        ghostDist += dist

```

```python
    '''
    Since capsules' score is very high, eating them all will lead to a higher score. Meanwhile, eating the ghost also
    gives us high score, so I add a 'hunt mode'. After the pacman eat the capsule, the pacman would turn into hunting
    mode. It would try to chase the ghost and eat it.
    '''
    capsule = currentGameState.getCapsules()
    capsuleNum = len(capsule)
    GhostStates = currentGameState.getGhostStates()
    hunt = False
    for state in GhostStates:
        if state.scaredTimer > 5:
            hunt = True

    '''
    I seperate the final sum into two conditions: one is hunt mode, the other is normal mode. First, in  normal mode,
    I put the score, minimum food distance, ghost's distance, number of capsules and the number of remaining food
    into consideration and calculate the total value. On the other hand, in hunt mode, the closer the ghost is, the
    higher chance the pacman can eat it, so the weight of 1/ghostDist is +10 compared to the one(-1) in normal mode.
    Also, in this mode, I lower the weight of the remaining food from 100 to 10, which means that the priority of
    eating the ghost is higher than eating food.
    '''
    score = currentGameState.getScore()
    if hunt:
        total = 1000 * score + 10 * (1 / minDist) + 10 * (1 / ghostDist) - 500 * capsuleNum - 200 * foodNum
    else:
        total = 1000 * score + 100 * (1 / minDist) - 1 * (1 / ghostDist) - 500 * capsuleNum - 200 * foodNum
    return total

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

Part II. Results & Analysis (5%):

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


                  ALL HAIL GRANDPAC.
           LONG LIVE THE GHOSTBUSTING KING.
```