Homework 3: Multi-Agent Search

Part I. Implementation (5%):

Part 1: Minimax Search

```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)
        """
            "minimaxScore" function is a function that returns the score when
            the game ends or the defined depth is reached.
            The function would maximize score for "Pacman" and minimize score for "ghosts".

            For Pacman's maximization part, the function should find the maximum
            score of the results obtained by running minimaxScore() with next agent,
            the current depth, and the child game state.

            Similarly, in the ghost minimization part, the function should find
            the minimum score with the same function and parameters.

            Finally, to perform the maximum action for Pacman, traverse
            through Pacman's legal moves and use minimaxScore() during the process.

        """
        def minimaxScore(agent, depth, gameState):
            # check whether game is ended or reaches the defined depth
            if gameState.isWin() or gameState.isLose() or depth == self.depth:
                return self.evaluationFunction(gameState) # return score

            # get next agent and update depth when all the agents have been traversed
            next_agent = agent + 1
            if gameState.getNumAgents() == next_agent:
                next_agent = 0
                depth += 1

            # if the agent is Pacman, return maximum score for all legal action of the agent
            if agent == 0:
                return max(minimaxScore(next_agent, depth, gameState.getNextState(agent, act)) for act in gameState.getLegalActions(agent))

            # if the agent is the ghost, return minimum score for all legal action of the agent
            else:
                return min(minimaxScore(next_agent, depth, gameState.getNextState(agent, act)) for act in gameState.getLegalActions(agent))

        maximum_score = float("-inf")
        move = random.choice(gameState.getLegalActions(0))
        # iterate all the Pacman's legal actions
        for act in gameState.getLegalActions(0):
            score = minimaxScore(1, 0, gameState.getNextState(0, act))

            # update score and move when score > maximum score
            if score > maximum_score:
                maximum_score = score
                move = act

        return move
        # End your code (Part 1)
```

Part 2: Alpha-Beta Pruning

```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)
        """
           The algorithm is similar to Minimax, but it uses two values, alpha (a)
           and beta (b), as thresholds to determine if pruning is necessary.

           Then, define an "alphabetaprune()" function that returns the score
           if the game ends or the defined depth is reached. The function should
           maximize or minimize Pacman or ghosts, just like in Minimax. Additionally,
           the function should use alpha (a) and beta (b) to determine if pruning is necessary.

           Finally, to perform the maximum action for the root (Pacman), traverse
           through Pacman's legal moves and use "alphabetaprune()" during the process.

        """

        def alphabetaprune(agent, depth, gameState, a, b):
            # check whether game is ended or reaches the defined depth
            if gameState.isWin() or gameState.isLose() or depth == self.depth:
                return self.evaluationFunction(gameState)

            # get next agent and update depth when all the agents have been traversed
            next_agent = agent + 1
            if gameState.getNumAgents() == next_agent:
                next_agent = 0
                depth += 1

            # if the agent is Pacman, find the maximum score for all legal action of the agent and prune the unnecessary branch
            if agent == 0:
                score = float("-inf")

                for act in gameState.getLegalActions(agent):

                    # find the maximum score of Pacman
                    score = max(score, alphabetaprune(next_agent, depth, gameState.getNextState(agent, act), a, b))

                    # prune the branch
                    if score > b:
                        return score

                    # update alpha
                    a = max(a, score)

                return score

            # if the agent is ghosts, find the minimum score for all legal action of the agent and prune the unnecessary branch
            else:
                score = float("inf")
                for act in gameState.getLegalActions(agent):

                    # find minimum score of the ghost
                    score = min(score, alphabetaprune(next_agent, depth, gameState.getNextState(agent, act), a, b))

                    # prune the branch
                    if score < a:
                        return score

                    # update beta
                    b = min(b, score)

                return score

        alpha = float("-inf")
        beta = float("inf")
        maximum_score = float("-inf")
        move = gameState.getLegalActions(0)[0]
        # iterate all the Pacman's legal actions
        for act in gameState.getLegalActions(0):

            score = alphabetaprune(1, 0, gameState.getNextState(0, act), alpha, beta)

            # update score and move when score > maximum score
            if score > maximum_score:
                maximum_score = score
                move = act

            # update alpha
            alpha = max(alpha, maximum_score)

        return move
        # End your code (Part 2)
```

Part 3: Expectimax Search

```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)
        """
            Similar to Minimax, define an "expectimax" function that returns the
            score if the game ends or the defined depth is reached. The function
            should maximize for Pacman when the "agent" is 0 but choose the branch
            by expected score for ghosts (chance) when the "agent" is not 0.

            Finally, to perform the maximum action for the root (Pacman), traverse
            through Pacman's legal moves and use "expectimax()" during the process.

        """
        def expectimax(agent, depth, gameState):
            # check whether game is ended or reaches the defined depth
            if gameState.isLose() or gameState.isWin() or depth == self.depth:
                return self.evaluationFunction(gameState)

            # get next agent and update depth when all the agents have been traversed
            next_agent = agent + 1
            if gameState.getNumAgents() == next_agent:
                next_agent = 0
                depth += 1

            # if the agent is Pacman, return maximum score for all legal action of the agent
            if agent == 0:
                return max(expectimax(next_agent, depth, gameState.getNextState(agent, act)) for act in gameState.getLegalActions(agent))

            # if the agent is ghost, return the expected score for all legal action of the agent
            else:
                return sum(expectimax(next_agent, depth, gameState.getNextState(agent, act)) for act in gameState.getLegalActions(agent)) / float(len(gameState.getLegalActions(agent)))

        maximum_score = float("-inf")
        move = gameState.getLegalActions(0)[0]
        # iterate all the Pacman's legal actions
        for act in gameState.getLegalActions(0):

            score = expectimax(1, 0, gameState.getNextState(0, act))

            # update score and move when score > maximum score
            if score > maximum_score:
                maximum_score = score
                move = act

        return move
        # End your code (Part 3)
```

Part 4: Evaluation Function

```python
def betterEvaluationFunction(currentGameState):
    """
    Your extreme ghost-hunting, pellet-nabbing, food-gobbling, unstoppable
    evaluation function (Part 4).
    """
    # Begin your code (Part 4)
    """
    This code defines an evaluation function for a Pacman game.
    It calculates a score for the current game state based on several game
    features, such as the current score, distance to the closest food, distance
    to the closest active ghost, and distance to the closest scared ghost. The
    function also takes into account the presence of capsules and the number of
    remaining food pellets. The game features are combined using a weighted
    linear combination to produce a final score. If Pacman loses the game,
    the score returned is negative infinity. If Pacman wins the game, the score
    returned is positive infinity.
    """

    # if Pacman lose the game, return the score as negative infinity
    if currentGameState.isLose():
        return float("-inf")

    # if Pacman win the game, return the score as positive infinity
    elif currentGameState.isWin():
        return float("inf")

    score = currentGameState.getScore()

    pacman_pos = currentGameState.getPacmanPosition()
    ghost_positions = currentGameState.getGhostPositions()
    distance_to_closest_active_ghost = float("inf")
    distance_to_closest_scared_ghost = float("inf")
    flg_active_ghost_too_close = 0
    flg_scared_ghost_too_close = 0

    food_list = currentGameState.getFood().asList()
    food_count = len(food_list)
    distance_to_closest_food = float("inf")

    capsules_count = len(currentGameState.getCapsules())

    # find the closest food distance for all the food left on the board
    food_distances = [manhattanDistance(pacman_pos, food_position) for food_position in food_list]
    if food_count > 0:
        distance_to_closest_food = min(food_distances)

    # a function to compute distance between the ghost and Pacman
    def getManhattanDistances(ghosts):
        return map(lambda g: util.manhattanDistance(pacman_pos, g.getPosition()), ghosts)

    # find all the scared ghosts and active ghost
    scared_ghosts, active_ghosts = [], []
    for ghost in currentGameState.getGhostStates():
        if not ghost.scaredTimer:
            active_ghosts.append(ghost)
        else:
            scared_ghosts.append(ghost)

    # compute the distance to the closest active ghost
    if active_ghosts:
        distance_to_closest_active_ghost = min(getManhattanDistances(active_ghosts))

    # if active ghost is too close, set the flag to be 1
    if distance_to_closest_active_ghost < 2:
        flg_active_ghost_too_close = 1

    # comput the distance to the closest scared ghost
    if scared_ghosts:
        distance_to_closest_scared_ghost = min(getManhattanDistances(scared_ghosts))

    # if scared ghost is too close, set the flag to be 1
    if distance_to_closest_scared_ghost < 2:
        flg_scared_ghost_too_close = 1

    game_feature = [score,
                    distance_to_closest_food,
                    1.0 / distance_to_closest_active_ghost,
                    flg_active_ghost_too_close,
                    1.0 / distance_to_closest_scared_ghost,
                    flg_scared_ghost_too_close,
                    capsules_count,
                    food_count]

    weight = [1,
              -1.5,
              -2,
              -100,
              20,
              100,
              -20,
              -4]

    # compute the final score as the linear combination of game features
    final_score = 0
    for i in range(len(game_feature)):
        final_score += game_feature[i] * weight[i]

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

Part II. Results & Analysis (5%):

- Result of autograder:

```
Average Score: 1222.2
Scores:        1146.0, 1076.0, 1121.0, 1273.0, 1305.0, 1308.0, 1354.0, 976.0, 1352.0, 1311.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
*** PASS: test_cases\part4\grade-agent.test (8 of 8 points)
*** EXTRA CREDIT: 2 points
***     1222.2 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 16:29:49

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

- Observation of my evaluation function:

```
game_feature = [score,
                distance_to_closest_food,
                1.0 / distance_to_closest_active_ghost,
                flg_active_ghost_too_close,
                1.0 / distance_to_closest_scared_ghost,
                flg_scared_ghost_too_close,
                capsules_count,
                food_count]

weight = [1,
          -1.5,
          -2,
          -100,
          20,
          100,
          -20,
          -4]
```

```
rule of score:
time pass: -1
eat food: +10
eat scared ghost: +200
win: +500
lose: -500
```

The game feature and weight are defined in the image above. Negative weight values indicate undesired states, while positive weight values indicate desired states. Based on this score calcu lation, my strategy is to prioritize killing scared ghosts. To achieve this, I assign a high weight to the "distance to the closest scared ghost" feature and the flag indicating the presence of a ne arby scared ghost.

Although my implementation can pass all the cases in autograder.py, there are still some limita tions to it. For example, if you run the code like "*python pacman.py -l smallClassic -p Expectima* *xAgent -a evalFn=better -n 10 -q*” , the result may not be optimal, as shown in the following ima ge:

```
Pacman emerges victorious! Score: 1144
Average Score: 555.0
Scores:        -381.0, -405.0, 1752.0, 1335.0, -136.0, -369.0, 1648.0, -382.0, 1344.0, 1144.0
Win Rate:      5/10 (0.50)
Record:        Loss, Loss, Win, Win, Loss, Loss, Win, Loss, Win, Win
```

However, it may sometimes achieve a decent score, as demonstrated in the following image:

```
Pacman emerges victorious! Score: 1511
Pacman emerges victorious! Score: 1685
Pacman emerges victorious! Score: 1550
Pacman emerges victorious! Score: 1529
Pacman emerges victorious! Score: 1521
Pacman emerges victorious! Score: 1704
Pacman emerges victorious! Score: 1678
Pacman emerges victorious! Score: 1741
Pacman emerges victorious! Score: 1348
Pacman emerges victorious! Score: 1703
Average Score: 1597.0
Scores:        1511.0, 1685.0, 1550.0, 1529.0, 1521.0, 1704.0, 1678.0, 1741.0, 1348.0, 1703.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

You can try to adjust the weight in the array to see whether you can get a better result.