HW3

111101013 蔡長頤

I. Code explanation:

1-1: Minimax Search:

- Base Case: If the game state is a win or lose state, or the maximum depth is reached, it evaluates the state using the evaluation function.
- Maximizing for Pac-Man: If agentIndex is 0 (Pac-Man's turn), it selects the action that maximizes the expected utility by recursively calling minimax for each legal action and returning the maximum value.

```
return max(minimax(state.getNextState(agentIndex, action), depth, 1) for action in state.getLegalActions(agentIndex)
```

- Minimizing for Ghosts: If agentIndex is not 0 (ghosts' turn), it selects the action that minimizes the expected utility by recursively calling minimax for each legal action and returning the minimum value.

```
return min(minimax(state.getNextState(agentIndex, action), nextDepth, nextAgent) for action in state.getLegalActions(agentIndex))
```

- Multi-agent problem:
  To cycle through each ghost and Pac-man, nextAgent is calculated in a matter that if the current agent is the last ghost, nextAgent will be 0 (Pac-Man's index).
  nextDepth is incremented only when all agents (including all ghosts) have had their turn, marking a new level in the game tree.

```
class MinimaxAgent(MultiAgentSearchAgent):
    def getAction(self, gameState):

        gameState.isLose():
        Returns whether or not the game state is a losing state
        """
        "*** YOUR CODE HERE ***"
        # Begin your code
        def minimax(state, depth, agentIndex):
            if state.isWin() or state.isLose() or depth == self.depth:
                return self.evaluationFunction(state)
            if agentIndex == 0:  # Pac-Man's move
                return max(minimax(state.getNextState(agentIndex, action), depth, 1) for action in state.getLegalActions(agentIndex))
            else:  # Ghosts' move
                nextAgent = agentIndex + 1 if agentIndex < gameState.getNumAgents() - 1 else 0
                nextDepth = depth + 1 if nextAgent == 0 else depth
                return min(minimax(state.getNextState(agentIndex, action), nextDepth, nextAgent) for action in state.getLegalActions(agentIndex))

        # Start the recursion from Pac-Man's turn, depth 0
        return max(gameState.getLegalActions(0), key=lambda x: minimax(gameState.getNextState(0, x), 0, 1))
```

1-2: Expectimax Search:

- Base Case: If the game state is a win or lose state, or the depth is 0, it evaluates the state using the evaluation function.
- Maximizing for Pac-Man: If agentIndex is 0 (Pac-Man's turn), it selects the action that maximizes the expected utility by recursively calling expectimax for each legal action and returning the maximum value.

```
return max(expectimax(state.getNextState(agentIndex, action), depth, 1) for action in state.getLegalActions(agentIndex))
```

- Expectation Step for Ghosts: If agentIndex is not 0 (ghosts' turn), it calculates the expected utility by averaging the values returned by recursively calling expectimax for each legal action.

```
return sum(expectimax(state.getNextState(agentIndex, action), depth, nextAgent) for action in actions) / len(actions)
```

```
# Begin your code
def expectimax(state, depth, agentIndex):
    if state.isWin() or state.isLose() or depth == 0:
        return self.evaluationFunction(state)

    if agentIndex == 0:  # Maximizer: Pac-Man
        return max(expectimax(state.getNextState(agentIndex, action), depth, 1) for action in state.getLegalActions(agentIndex))
    else:  # Expectation: Ghosts
        nextAgent = agentIndex + 1
        if nextAgent >= state.getNumAgents():
            nextAgent = 0
            depth -= 1
        actions = state.getLegalActions(agentIndex)
        return sum(expectimax(state.getNextState(agentIndex, action), depth, nextAgent) for action in actions) / len(actions)

# Start from Pac-Man's perspective
maximum_value = float("-inf")
action_to_take = None
for action in gameState.getLegalActions(0):
    value = expectimax(gameState.getNextState(0, action), self.depth, 1)
    if value > maximum_value:
        maximum_value = value
        action_to_take = action
return action_to_take
# End your code
```

2-1: Value iteration:

- Value iteration looping:
  - State Values Update:

    For each state in the MDP (Markov Decision Process), the code checks if the state is terminal. If it is, the value for that state is set to 0 since terminal states have no future rewards.For non-terminal states, the code calculates the maximum expected value for the state using the Bellman equation.
  - Value Calculation for Actions:

    The Bellman equation calculates the expected value for each state as the maximum value across all possible actions, where the value for each action is the sum of the immediate reward and the discounted future value of the next state.

$$V_{opt}^{(t)}(s) \leftarrow Argmax_{a \in Action(s)} \sum_{s'} T(s,a,s')[Reward(s,a,s') + \gamma V_{opt}^{(t-1)}(s')]$$

```
for action in self.mdp.getPossibleActions(state):
    total = sum(prob * (self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState])
                for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action))
    max_value = max(max_value, total)
new_values[state] = max_value
```

```
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    # Begin your code
    for i in range(self.iterations):
        new_values = {}
        for state in self.mdp.getStates():
            if self.mdp.isTerminal(state):
                new_values[state] = 0
            else:
                max_value = float('-inf')
                for action in self.mdp.getPossibleActions(state):
                    total = sum(prob * (self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState])
                                for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action))
                    max_value = max(max_value, total)
                new_values[state] = max_value
        self.values = new_values
    # End your code
```

- Compute Q values:

  The loop corresponds to the equation below:

$$Q_\pi(s,a) = \sum_{s'} T(s,a,s')[Reward(s,a,s') + \gamma V_\pi(s')]$$

```
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    return sum(prob * (self.mdp.getReward(state, action, nextState) + self.discount * self.values[nextState])
               for nextState, prob in self.mdp.getTransitionStatesAndProbs(state, action))
    # End your code
```

- Compute the best action:
  This function returns the best action(highest Q value) at a given state.

```
def computeActionFromValues(self, state):
    """
      The policy is the best action in the given state
      according to the values currently stored in self.values.

      You may break ties any way you see fit.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return None.
    """
    "*** YOUR CODE HERE ***"
    # Begin your code

    if self.mdp.isTerminal(state):
        return None
    return max(self.mdp.getPossibleActions(state), key=lambda action: self.getQValue(state, action))

    # End your code
```

2-2, 2-3: Q-Learning:

- Initialize Q-values: Start with zero values for all state-action pairs.

```
def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)

    "*** YOUR CODE HERE ***"
    # Begin your code
    self.qValues = {}          # Dictionary to hold Q-values, defaulting to 0
    # End your code
```

- Epsilon-greedy action selection:
  Use an ε-greedy policy for action selection to balance exploration and exploitation.
  - Exploration:
    Handled by random.choice(legalActions) when flipCoin(self.epsilon) returns True.
  - Exploitation:
    Handled by computeActionFromQValues(state) when flipCoin(self.epsilon) returns False. Ties are broken randomly by choosing one of the actions that have the highest Q-value.

```
def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.

      HINT: You might want to use util.flipCoin(prob)
      HINT: To pick randomly from a list, use random.choice(list)
    """
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None
    "*** YOUR CODE HERE ***"
    # Begin your code
    if flipCoin(self.epsilon):  # Explore
        return random.choice(legalActions)
    else:  # Exploit
        q_values = [self.getQValue(state, a) for a in legalActions]
        max_q_value = max(q_values)

        return random.choice([a for a, q in zip(legalActions, q_values) if q == max_q_value])
    # End your code
```

- Update Q-values: After each action, update the Q-values based on the received reward and the maximum estimated future reward. The code below corresponds to the equation:

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a'} Q(s',a') - Q(s,a)]$$

```
def update(self, state, action, nextState, reward):
    """
      The parent class calls this to observe a
      state = action => nextState and reward transition.
      You should do your Q-Value update here

      NOTE: You should never call this function,
      it will be called on your behalf
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    legalActions = self.getLegalActions(nextState)
    if legalActions:  # Check if there are any legal actions available from nextState
        next_max_q = max([self.getQValue(nextState, a) for a in legalActions])
    else:
        next_max_q = 0  # If no legal actions, next_max_q should be 0 (terminal state)

    old_q_value = self.getQValue(state, action)
    new_q_value = old_q_value + self.alpha * (reward + self.discount * next_max_q - old_q_value)
    self.qValues[(state, action)] = new_q_value

    # End your code
```

2-4: Approximate Q-learning;

Approximate Q-learning is generally similar to Q-learning, except that we have feature representation and weight updates.

- Feature representation: Instead of having a Q-value for each state-action pair, the agent learns weights for features that describe these pairs.
  Besides the given features in the code, I added two additional features: “Distance to the Nearest Ghost” and “Scared Ghosts Edible Time” to capture the nuances of the game better.

```
##########
    #distance to the nearest ghost
    from util import manhattanDistance
    ghostDistances = [manhattanDistance((next_x, next_y), ghostPos) for ghostPos in ghosts]
    features['ghost-distance-min'] = min(ghostDistances) / (walls.width * walls.height)
##########
    #Scared Ghosts and Edible Time
    ghostState = state.getGhostStates()
    scaredTimes = [ghost.scaredTimer for ghost in ghostState]

    # Count the number of scared ghosts
    features["scared-ghosts-count"] = sum(1 for time in scaredTimes if time > 0)

    # Normalize the total scared time remaining
    if features["scared-ghosts-count"] > 0:
        features["total-scared-time"] = sum(scaredTimes) / len(scaredTimes)
    else:
        features["total-scared-time"] = 0
##########
```

A slight improvement in the autograder is observed.

```
865.66 average score (2 of 4 points)
    Grading scheme:
     < 500:  0 points
    >= 500:  2 points
    >= 1000:  4 points
100 games not timed out (2 of 2 points
    Grading scheme:
     < 0:  fail
    >= 0:  0 points
    >= 5:  1 points
    >= 10:  2 points
90 wins (3 of 4 points)
    Grading scheme:
     < 1:  fail
    >= 1:  1 points
    >= 40:  2 points
    >= 70:  3 points
    >= 100:  4 points
```

```
886.9 average score (2 of 4 points)
    Grading scheme:
     < 500:  0 points
    >= 500:  2 points
    >= 1000:  4 points
100 games not timed out (2 of 2 points)
    Grading scheme:
     < 0:  fail
    >= 0:  0 points
    >= 5:  1 points
    >= 10:  2 points
92 wins (3 of 4 points)
    Grading scheme:
     < 1:  fail
    >= 1:  1 points
    >= 40:  2 points
    >= 70:  3 points
    >= 100:  4 points
```

▲Performance of Original feature extractor(left) and Modified feature extractor(right)

- Q-value calculations:
  Q-values are computed as the dot product of the features and their corresponding weights.

```
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    # get weights and feature
    featureVector = self.featExtractor.getFeatures(state, action)
    return sum(self.weights[feature] * value for feature, value in featureVector.items())
    # End your code
```

- Weight Update: The weights are updated based on the difference between the estimated Q-values and the observed rewards, similar to standard Q-learning but applied to the feature weights.

```
def update(self, state, action, nextState, reward):
    """
      Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    # Begin your code
    prediction = self.getQValue(state, action)
    target = reward + self.discount * self.getValue(nextState)
    correction = (target - prediction)
    featureVector = self.featExtractor.getFeatures(state, action)

    for feature, value in featureVector.items():
        self.weights[feature] += self.alpha * correction * value
    # End your code
```

II. Questions:

1. What is the difference between On-policy and Off-policy.
   - On-policy methods learn the value of the policy being carried out by the agent, including the exploration steps. The policy directs the agent's actions in every state, including the decision-making process while learning. The agent evaluates the outcomes of its present actions, refining its strategy incrementally. An example is SARSA (State-Action-Reward-State-Action), where the policy learned is derived from the actions taken by the current policy.
   - Off-policy methods learn the value of the optimal policy, regardless of the agent's actions. This allows the agent to learn from actions that are outside the current policy, such as using a greedy policy while following a random policy. Q-learning is a popular off-policy method where the learning is based on the assumption of selecting the best available action.
2. Briefly explain value-based, policy-based and Actor-Critic. Also, describe the value function $V^{\pi}(S)$.
   - Value-based methods focus on finding the value function which tells us the long-term reward for each state, helping the agent to decide which action to take. The agent uses this value function to select actions that maximize the expected reward.
   - Policy-based methods directly learn the policy function that maps state to action without using a value function as an intermediary. These methods optimize the policy by using gradients to find the best actions.
   - Actor-Critic methods combine both value-based and policy-based methods. The "actor" updates the policy distribution in the direction suggested by the "critic," which evaluates the action taken by the actor by computing the value function.

- Value Function $V^{\pi}(S)$is the expected return starting from state S and following policy π thereafter. Essentially, it provides a prediction of future rewards given a state and a policy.

3. What is the difference between Monte-Carlo (MC) based approach and Temporal-difference (TD) approach for estimating $V^{\pi}(S)$.
   - Monte-Carlo methods estimate the value function by averaging the returns of complete episodes (data). These methods do not bootstrap (i.e., they do not update their estimate based on other estimates) but rely solely on full episodes(data), making them suitable for episodic tasks.
   - Temporal-Difference methods learn directly from incomplete episodes by bootstrapping. They update their estimates based on the combination of learned estimates and data without waiting for a final outcome. This makes TD methods generally faster and applicable to continuous tasks as well as episodic tasks.
   - For example, in TD learning, **w** is updated with itself, **r**, which is from data, and $\widehat{\boldsymbol{V}}_{\boldsymbol{\pi}}$, which is the estimate. In contrast, in model free Monte Carlo, $\widehat{\boldsymbol{Q}_{\boldsymbol{\pi}}}$ is updated with itself and solely **u** of each (s, a, u).
4. Describe State-action value function $Q^{\pi}(s, a)$ and the relationship between $V^{\pi}(S)$ in Q-learning.
   The State-action value function $Q^{\pi}(s, a)$ estimates the expected return starting from state s, taking action a, and thereafter following a policy π. It helps in determining the best action to take in a given state.
   $V^{\pi}(S)$ is derived from Q-values by $V^{\pi}(S) = \max_a Q^{\pi}(s, a)$, meaning the value of being in a state s under policy π is the maximum Q-value achievable from that state.
5. Describe following tips Target Network, Exploration and Replay Buffer using in Q-learning.
   - Target Network: In deep Q-learning, a separate network is used to estimate the target values that stabilize training. This network has the same architecture as the primary network but its weights are updated less frequently.
   - Exploration: Essential in Q-learning to avoid local minima and ensure a good policy across all states. Common strategies include ε-greedy, where most actions are chosen by the current best known policy, but with a probability ε, a random action is chosen.
   - Replay Buffer: Stores the agent's experiences, which are tuples of (state, action, reward, next state). The buffer allows the agent to reuse past experiences in a random order, breaking the correlation between sequential experiences and stabilizing learning.
6. Explain what is different between DQN and Q-learning.
   - Q-learning is an approach that updates the Q-values for each state-action pair directly.
   - Deep Q-Networks (DQN) extend Q-learning to large state spaces by using a neural network to approximate the Q-value function. DQN addresses the challenges of stability and convergence in high-dimensional spaces through techniques like experience replay and fixed target networks.

III. Compare the performance of every method and do some discussions in your report.

1. Minimax vs Expectimax:
   In the case of smallClassic, Minimax and Expectimax performed identical. However, in the case of trappedClassic, Expectimax performed significantly greater than Minimax. This implies that when the ghosts have a nature of uncertainty and don't always try to minimize things, Minimax might lead to overly conservative strategies because it always assumes the worst-case scenario. On the other hand, Expectimax takes different

probabilities of various outcomes into account, which is more realistic in scenarios where opponents may make mistakes or where chance plays a role.

```
*** Running MinimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running MinimaxAgent on smallClassic after 20 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part1-1\8-pacman-game.test

### Question part1-1: 10/10 ###
```

▲Minimax, smallClassic

```
*** Running ExpectimaxAgent on smallClassic 1 time(s).
Pacman died! Score: 84
Average Score: 84.0
Scores:        84.0
Win Rate:      0/1 (0.00)
Record:        Loss
*** Finished running ExpectimaxAgent on smallClassic after 20 seconds.
*** Won 0 out of 1 games. Average score: 84.000000 ***
*** PASS: test_cases\part1-2\7-pacman-game.test

### Question part1-2: 10/10 ###
```

▲Expectimax, smallClassic

```
PS C:\Users\yvonn\OneDrive\Desktop\交大\人工智慧概論\HW\HW3\HW3\HW3_\Adversarial_search> python pacman.py -p MinimaxAgent -l trappedClassic -a depth=3 -q -n 10
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Pacman died! Score: -501
Average Score: -501.0
Scores:        -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0, -501.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

▲Minimax, trappedClassic

```
PS C:\Users\yvonn\OneDrive\Desktop\交大\人工智慧概論\HW\HW3\HW3\HW3_\Adversarial_search> python pacman.py -p ExpectimaxAgent -l trappedClassic -a depth=3 -q -n 10
Pacman emerges victorious! Score: 532
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Pacman died! Score: -502
Pacman emerges victorious! Score: 532
Pacman died! Score: -502
Pacman died! Score: -502
Average Score: -88.4
Scores:        532.0, 532.0, -502.0, 532.0, -502.0, -502.0, -502.0, 532.0, -502.0, -502.0
Win Rate:      4/10 (0.40)
Record:        Win, Win, Loss, Win, Loss, Loss, Loss, Win, Loss, Loss
```

▲Expectimax, trappedClassic

2. The effect of epsilon on Epsilon-greedy action selection

Ɛ = 0.1	Ɛ = 0.5	Ɛ = 1.0
Got reward: 1 EPISODE 100 COMPLETE: RETURN WAS 0.5904900000000002 AVERAGE RETURNS FROM START STATE: 0.4772159588556872	Got reward: 1 EPISODE 100 COMPLETE: RETURN WAS 0.38742048900000015 AVERAGE RETURNS FROM START STATE: 0.3191918136765479	Got reward: 1 EPISODE 100 COMPLETE: RETURN WAS 0.00024274944503154723 AVERAGE RETURNS FROM START STATE: -0.08703735404920357

As we can see, when Ɛ = 1.0, there is a significantly lower return, and a negative average return from the start state of about -0.087. Therefore, we can say that in this specific task, lower epsilon values (like 0.1) performed better, because it leads to strategies that exploit more than explore. The algorithm primarily sticks with the best-known actions, which can be effective if the value estimates are accurate but could also miss out on potentially better actions not yet explored. Higher epsilon values (like 0.9) lead to a lot of exploration. This can be useful in the early stages of learning or in non-stationary environments where the optimal strategy changes over time. However, it can also lead to suboptimal short-term performance and slower convergence because the algorithm spends less time exploiting what it has learned.

3. Comparison of value iteration and Q-learning:

- Convergence rate:
  When I am writing the homework, it is obvious that value iteration has a higher convergence speed than Q-learning. Value Iteration may converge faster in terms of the number of iterations because it updates all states based on a full model of the

environment. Q-Learning may require more interactions to achieve convergence, especially in large or complex state spaces, because it weights the prediction and the target, considering things more sophisticatedly.

4. Comparison between Pacman Q agent and Approximate Q Agent on smallGrid:

In this comparison, Approximate Q agent performs worse than Pacman Q agent. Possible reasons may be:

- Insufficient Training: The Approximate Q-learning agent is trained with significantly fewer episodes (50) compared to the Q-learning agent (2000). This considerable difference in the amount of training can result in the Approximate Q-learning agent not learning an effective policy, especially if the feature extractor used does not capture all relevant aspects of the state space effectively.
- Generalization Issues: Approximate Q-learning aims to generalize across similar states, which can be an advantage in large state spaces. However, in smaller or simpler environments like smallGrid, this generalization might not be necessary or could even be detrimental if it smooths over important distinctions between states.

```
Average Score: 501.4
Scores:        503.0, 501.0, 495.0, 503.0, 503.0, 499.0, 501.0, 503.0, 503.0, 503.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

▲Pacman Q agent after 2000 trainings, and a total of 2010 games on smallGrid

```
Average Score: 299.6
Scores:        507.0, 507.0, -504.0, 495.0, 499.0, 495.0, 495.0, 499.0, 507.0, -504.
Win Rate:      8/10 (0.80)
Record:        Win. Win. Loss. Win. Win. Win. Win. Win. Win. Loss
```

▲Approximate Q agent after 50 trainings, and a total of 60 games on smallGrid

5. Comparison between Pacman Q agent, Approximate Q Agent, and DQN Agent on smallClassic:

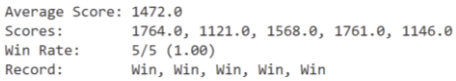
- Pacman Q Agent:
  This agent did not win any games. This performance could be due to insufficient training (only 2000 training episodes might not be enough), and mainly the fact that the complexity of the smallClassic (which is a lot more complex than smallGrid) is beyond the capability Q-learning might could generalize.
- Approximate Q Agent:
  Despite only 50 training episodes, the Approximate Q agent has performed significantly better than the basic Pacman Q agent. This suggests that the features used by the Approximate Q agent were effective at capturing the important aspects of the game environment, allowing it to generalize better and learn a reasonably good policy with far less training.
- DQN Agent:
  The significantly higher average score indicates that the DQN has learned a much more effective strategy for the game. The DQN agent benefits from experience replay and target networks, which stabilize learning and allow for a better generalization of the policy.

```
Average Score: -392.9
Scores:        -366.0, -419.0, -430.0, -386.0, -390.0, -409.0, -341.0, -397.0, -406.0, -385.0
Win Rate:      0/10 (0.00)
Record:        Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss, Loss
```

▲Pacman Q agent on smallClassic

```
Average Score: 216.7
Scores:        -73.0, 960.0, 836.0, -310.0, -353.0, 925.0, -273.0, -174.0, -263.0, 892.0
Win Rate:      4/10 (0.40)
Record:        Loss, Win, Win, Loss, Loss, Win, Loss, Loss, Loss, Win
```

▲Approximate Q agent on smallClassic



▲DQN agent on smallClassic

IV. Describe problems you meet and how you solve them.

1. The main problem I encountered in the beginning of the homework is to understand the configurations and provided functions in multiple files.
2. Scoring in Autograder:
   My Approximate Q Agent scored 7/10 in the autograder, and I wanted to improve its performance. I identified specific areas for improvement in my feature extractor, such as considering scared ghosts, edible time, and power pellet proximity. By enhancing my feature extractor to capture these additional aspects of the game state, I improved my agent's performance, but still couldn’t meet the next level of grading and get more points.