گزارش پروژه سوم

اشکان شکیبا (9931030)

بخش اول

```
class ValueIterationAgent(ValueEstimationAgent):
    """
    * Please read learningAgents.py before reading this.*

    A ValueIterationAgent takes a Markov decision process
    (see mdp.py) on initialization and runs value iteration
    for a given number of iterations using the supplied
    discount factor.
    """

    def __init__(self, mdp, discount=0.9, iterations=100):
        """
        Your value iteration agent should take an mdp on
        construction, run the indicated number of iterations
        and then act according to the resulting policy.

        Some useful mdp methods you will use:
            mdp.getStates()
            mdp.getPossibleActions(state)
            mdp.getTransitionStatesAndProbs(state, action)
            mdp.getReward(state, action, nextState)
            mdp.isTerminal(state)
        """
        self.mdp = mdp
        self.discount = discount
        self.iterations = iterations
        self.values = util.Counter()  # A Counter is a dict with default 0
        self.runValueIteration()

    def runValueIteration(self):
        # Write value iteration code here

        iterations = self.iterations
        while iterations:
            current_value = util.Counter()
            for state in self.mdp.getStates():
                if not self.mdp.isTerminal(state):
                    max_value = -sys.maxsize
```

```
                    for action in self.mdp.getPossibleActions(state):
                        action_value = 0
                        for next_state, probability in
self.mdp.getTransitionStatesAndProbs(state, action):
                            reward = self.mdp.getReward(state, action,
next_state)
                            action_value += probability * (reward +
self.discount * self.values[next_state])
                        if action_value > max_value:
                            max_value = action_value
                    current_value[state] = max_value
            self.values = current_value
            iterations -= 1

    def getValue(self, state):
        """
        Return the value of the state (computed in __init__).
        """
        return self.values[state]

    def computeQValueFromValues(self, state, action):
        """
        Compute the Q-value of action in state from the
        value function stored in self.values.
        """

        action_value = 0
        for next_state, probability in
self.mdp.getTransitionStatesAndProbs(state, action):
            reward = self.mdp.getReward(state, action, next_state)
            action_value += probability * (reward + self.discount *
self.values[next_state])
        return action_value

    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.
        """

        if self.mdp.isTerminal(state):
            return None
        else:
            max_value = -sys.maxsize
            best_action = self.mdp.getPossibleActions(state)[0]
            for action in self.mdp.getPossibleActions(state):
                Q_value = self.computeQValueFromValues(state, action)
                if Q_value > max_value:
                    max_value = Q_value
                    best_action = action
```

```
            return best_action

    def getPolicy(self, state):
        return self.computeActionFromValues(state)

    def getAction(self, state):
        "Returns the policy at the state (no exploration)."
        return self.computeActionFromValues(state)

    def getQValue(self, state, action):
        return self.computeQValueFromValues(state, action)
```

در تابع runValueIteration فرایند value iteration انجام شده و با محاسبه ارزش هر عمل در هر حالت، ماکسیمم ارزش هر حالت تعیین و ذخیره می‌شود.

در تابع computeQValueFromValues مقادیر Q value به ازای هر عمل در هر حالت و با استفاده از ارزش حالت بعدی محاسبه و بازگردانی می‌شوند.

در تابع computeActionFromValues با در نظر گرفتن ارزش‌های حالت‌ها، عمل متناسب برای هر حالت انتخاب و بازگردانی می‌شود تا سیاست کلی پاسخ تعیین شود.

خروجی برنامه پس از ۱۰ iteration، که مطابق انتظار است.

بخش دوم

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0.015
    return answerDiscount, answerNoise
```

با کاهش نویز از ۰٫۲ به ۰٫۰۱۵ در واقع محیط را قطعی‌تر می‌کنیم و با این تغییر و بهینه شدن سیاست ما، عامل تلاش می‌کند از پل رد شود.

بخش سوم

```python
def question3a():
    answerDiscount = 0.5
    answerNoise = 0.0
    answerLivingReward = -5.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'


def question3b():
    answerDiscount = 0.5
    answerNoise = 0.1
    answerLivingReward = -1.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'


def question3c():
    answerDiscount = 1.0
    answerNoise = 0.0
    answerLivingReward = -1.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'


def question3d():
    answerDiscount = 1.0
    answerNoise = 0.1
    answerLivingReward = -0.1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'


def question3e():
    answerDiscount = 1.0
    answerNoise = 0.1
    answerLivingReward = 9999.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

در بخش a، پاداش زندگی عددی منفی و قابل توجه است تا عامل تلاش کند هر چه سریع‌تر بازی را به پایان برساند، پس به سراغ خروجی نزدیک‌تر می‌رود. ضمنا نویز نیز صفر است و عامل خطر صخره را می‌پذیرد.

در بخش b، مقدار نویز را افزایش می‌دهیم تا عامل به ریسک صخره اهمیت دهد و مسیر طولانی‌تر اما کم‌خطرتر را انتخاب کند.

در بخش c، برعکس بخش a مقدار پاداش زندگی طوری انتخاب می‌شود که عامل اهداف دورتر اما پرارزش‌تر را ترجیح دهد. همچنین مشابه بخش a نویز را صفر در نظر می‌گیریم تا عامل خطر صخره را بپذیرد.

در بخش d، مشابه بخش b و با افزایش مقدار نویز به عامل درباره ریسک صخره هشدار می‌دهیم و تغییر تخفیف به ۱/۰ عامل را تشویق به انتخاب اهداف دورتر اما پرارزش‌تر می‌کند.

در بخش e، پاداش زندگی را عددی بسیار بزرگ قرار می‌دهیم تا عامل همواره بازی را ادامه دهد و از پایان بازی در خروجی‌ها یا صخره‌ها بپرهیزد.

سوال)

نه لزوما، تنها زمانی که گاما عددی بین صفر و یک باشد این اتفاق می‌افتد، چرا که با افزایش تعداد دفعات و ضرب پیاپی آن، حاصل به سوی صفر میل می‌کند.

بخش چهارم

```
class AsynchronousValueIterationAgent(ValueIterationAgent):
    """
    * Please read learningAgents.py before reading this.*

    An AsynchronousValueIterationAgent takes a Markov decision process
    (see mdp.py) on initialization and runs cyclic value iteration
    for a given number of iterations using the supplied
    discount factor.
    """

    def __init__(self, mdp, discount=0.9, iterations=1000):
        """
        Your cyclic value iteration agent should take an mdp on
        construction, run the indicated number of iterations,
        and then act according to the resulting policy. Each iteration
        updates the value of only one state, which cycles through
        the states list. If the chosen state is terminal, nothing
        happens in that iteration.

        Some useful mdp methods you will use:
            mdp.getStates()
            mdp.getPossibleActions(state)
            mdp.getTransitionStatesAndProbs(state, action)
            mdp.getReward(state)
            mdp.isTerminal(state)
        """
        ValueIterationAgent.__init__(self, mdp, discount, iterations)

    def runValueIteration(self):
        iterations = self.iterations
        states = self.mdp.getStates()
        for iteration in range(iterations):
            state_index = iteration % len(states)
            state = states[state_index]
            if not self.mdp.isTerminal(state):
                max_value = -sys.maxsize
                for action in self.mdp.getPossibleActions(state):
                    action_value = 0
                    for next_state, probability in
self.mdp.getTransitionStatesAndProbs(state, action):
                        reward = self.mdp.getReward(state, action,
next_state)
                        action_value += probability * (reward +
self.discount * self.values[next_state])
                    if action_value > max_value:
                        max_value = action_value
                self.values[state] = max_value
```

در تابع runValueIteration، درون حلقه‌ای هر بار یک حالت بروزرسانی می‌شود؛ به این شکل که در اولین تکرار تنها مقدار حالت اول را بروز کرده و تا انجام فرایند برای همه حالت‌ها این روند ادامه می‌یابد. پس از این از حالت اول برای تکرارهای پس از آن استفاده می‌شود. در نهایت با یافتن بیشینه ارزش حاصل از عمل‌ها، مقدار ارزش حالت ذخیره می‌شود.

سوال)

بروزرسانی با استفاده از batch:

- نکته مثبت: مقادیر ارزش‌های حالات زودتر به همگرایی می‌رسند و همچنین مقداری که در پایان برای هر حالت محاسبه می‌شود به مقدار همگرایی آن نزدیک‌تر است.
- نکته منفی: نیاز به پردازش و صرف زمان بیشتر

بروزرسانی به صورت تکی:

- نکته مثبت: با صرف زمان و پردازش بسیار کمتر، عامل به دید نسبتا خوبی می‌رسد
- نکته منفی: مقادیر ارزش‌های حالات دیرتر به همگرایی می‌رسند و نسبت به حالت batch، تفاوت بیشتری با مقدار همگرایی خود دارند.

بخش پنجم

```
class
PrioritizedSweepingValueIterationAgent(AsynchronousValueIterationAgent):
    """
    * Please read learningAgents.py before reading this.*

    A PrioritizedSweepingValueIterationAgent takes a Markov decision
process
    (see mdp.py) on initialization and runs prioritized sweeping value
iteration
    for a given number of iterations using the supplied parameters.
    """

    def __init__(self, mdp, discount=0.9, iterations=100, theta=1e-5):
        """
        Your prioritized sweeping value iteration agent should take an mdp
on
        construction, run the indicated number of iterations,
        and then act according to the resulting policy.
        """
        self.theta = theta
        ValueIterationAgent.__init__(self, mdp, discount, iterations)

    def runValueIteration(self):
        predecessors = dict()
        states = self.mdp.getStates()

        for state in states:
            predecessors[state] = set()

        for state in states:
            actions = self.mdp.getPossibleActions(state)
            for action in actions:
                for next_state, probability in
self.mdp.getTransitionStatesAndProbs(state, action):
                    if probability != 0:
                        predecessors[next_state].add(state)

        queue = util.PriorityQueue()

        for state in states:
            if not self.mdp.isTerminal(state):
                Q_values = list()
                max_Q_value = -sys.maxsize
                current = self.values[state]
                for action in self.mdp.getPossibleActions(state):
                    Q_value = self.computeQValueFromValues(state, action)
                    Q_values.append(Q_value)
                    if Q_value > max_Q_value:
                        max_Q_value = Q_value
```

```
                if current > max_Q_value:
                    queue.update(state, max_Q_value - current)
                else:
                    queue.update(state, current - max_Q_value)

        for i in range(self.iterations):
            if queue.isEmpty():
                return

            state = queue.pop()
            if not self.mdp.isTerminal(state):
                values = list()
                max_value = -sys.maxsize
                for action in self.mdp.getPossibleActions(state):
                    value = 0
                    for next_state, probability in
self.mdp.getTransitionStatesAndProbs(state, action):
                        reward = self.mdp.getReward(state, action,
next_state)
                        next_Q_value = self.values[next_state]
                        value += probability * (reward + self.discount *
next_Q_value)
                    values.append(value)
                    if value > max_value:
                        max_value = value
                self.values[state] = max_value

            for previous in predecessors[state]:
                Q_values = list()
                max_Q_value = -sys.maxsize
                current = self.values[previous]
                for action in self.mdp.getPossibleActions(previous):
                    Q_value = self.computeQValueFromValues(previous,
action)
                    Q_values.append(Q_value)
                    if Q_value > max_Q_value:
                        max_Q_value = Q_value

                if abs(current - max_Q_value) > self.theta:
                    queue.update(previous, -abs(current - max_Q_value))
```

ابتدا پس از تعریف شدن predecessor برای هر حالت، بیشینه مقدار Q value محاسبه شده و Q value واقعی پیدا شده و منفی آن به صف اولویت افزوده می‌شود. سپس با خروج یک مقدار از صف، در صورتی که حالت ترمینال نباشد، بروزرسانی شده و با بررسی همه predecessorهای آن، اگر اختلاف Q value محاسبه شده حالت و predecessor از مقدار تتا بیشتر شود، منفی آن به صف اولویت افزوده می‌شود.

بخش ششم

```
class QLearningAgent(ReinforcementAgent):
    """
    Q-Learning Agent

    Functions you should fill in:
      - computeValueFromQValues
      - computeActionFromQValues
      - getQValue
      - getAction
      - update

    Instance variables you have access to
      - self.epsilon (exploration prob)
      - self.alpha (learning rate)
      - self.discount (discount rate)

    Functions you should use
      - self.getLegalActions(state)
        which returns legal actions for a state
    """

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

        self.qvalue = util.Counter()

    def getQValue(self, state, action):
        """
        Returns Q(state,action)
        Should return 0.0 if we have never seen a state
        or the Q node value otherwise
        """

        return self.qvalue[(state, action)]

    def computeValueFromQValues(self, state):
        """
        Returns max_action Q(state,action)
        where the max is over legal actions.  Note that if
        there are no legal actions, which is the case at the
        terminal state, you should return a value of 0.0.
        """

        if not self.getLegalActions(state):
            return 0.0
        max_Q_value = -sys.maxsize
        for action in self.getLegalActions(state):
            if self.qvalue[(state, action)] > max_Q_value:
                max_Q_value = self.qvalue[(state, action)]
```

```
        return max_Q_value

    def computeActionFromQValues(self, state):
        """
        Compute the best action to take in a state.  Note that if there
        are no legal actions, which is the case at the terminal state,
        you should return None.
        """

        if self.getLegalActions(state):
            return max([action for action in self.getLegalActions(state)],
key=lambda x: self.qvalue[(state, x)])

    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 ***"
        util.raiseNotDefined()

        return action

    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
        """

        self.qvalue[(state, action)] = self.qvalue[(state, action)] * (1 -
self.alpha) + \
                                       self.alpha *
(self.getValue(nextState) * self.discount + reward)

    def getPolicy(self, state):
        return self.computeActionFromQValues(state)

    def getValue(self, state):
        return self.computeValueFromQValues(state)
```

در تابع computeValueFromQValues با محاسبه بیشینه Q valueهای هر حالت، ارزش آن به دست می‌آید.

در تابع computeActionFromQValues مشابه تابع قبل بیشینه Q valueها محاسبه شده و عمل مربوط به ماکسیمم بازگردانی می‌شود.

در تابع update با استفاده از رابطه زیر، مقادیر Q valueها بروزرسانی می‌شود.

$$sample = R(s, a, a') + \gamma \max_a Q(s', a')$$

$$Q(s, a) \leftarrow (1 - \alpha) * Q(s, a) + \alpha * [sample]$$

سوال)

اگر مقدار Q برای اقداماتی که عامل قبلا ندیده زیاد باشد، عامل تمایل بیشتری به exploration محیط دارد و اگر مقدار آن کم باشد، عامل تمایل بیشتری به exploitation دارد.

این موضوع را می‌توان اینطور توضیح داد که در حالت اول عامل به سمت کارهایی که تاکنون انجام نداده می‌رود اما در حالت دوم ترجیح می‌دهد کارهای قبلی خود را تکرار کند.

بخش هفتم

```
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

    if not legalActions:
        return None
    if flipCoin(self.epsilon):
        return random.choice(legalActions)
    action = self.getPolicy(state)

    return action
```

در این تابع یا بهترین عمل شناخته شده تاکنون برگردانده می‌شود و یا یک حرکت تصادفی، که اولی در جهت exploitation است و دومی در جهت exploration؛ احتمال انتخاب هر کدام هم به مقدار اپسیلون بستگی دارد.

بخش هشتم

```
def question8():
    answerEpsilon = None
    answerLearningRate = None
    return "NOT POSSIBLE"
    # If not possible, return 'NOT POSSIBLE'
```

برای دستیابی به سیاست بهینه با احتمال بیشتر از ۹۹ درصد، ۵۰ اپیزود کم است و نیاز به اپیزودهای بیشتری داریم، چرا که لازمه این موضوع عبور از پل است که آسان نیست؛ بنابراین این تابع عبارت NOT POSSIBLE را بازگردانی می‌کند که به معنای امکان‌پذیر نبودن این امر است.

سوال)

افزایش اپسیلون منجر به تمایل عامل به exploration می‌شود و ترجیح می‌دهد کارهای جدید بیشتری را امتحان کند.

کاهش اپسیلون منجر به تمایل عامل به exploitation می‌شود و ترجیح می‌دهد کارهای نتیجه‌بخش قبلی را تکرار کند.

بخش نهم

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

با اجرای دستور بالا می‌بینیم که پس از ۲۰۰۰ اپیزود یادگیری، عامل موفق به برد در هر ۱۰ بازی می‌شود.

نتیجه اجرا:

```
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Average Score: 500.2
Scores:        499.0, 503.0, 503.0, 499.0, 495.0, 495.0, 503.0, 503.0, 503.0, 499.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

بخش دهم

```
class ApproximateQAgent(PacmanQAgent):
    """
    ApproximateQLearningAgent

    You should only have to overwrite getQValue
    and update.  All other QLearningAgent functions
    should work as is.
    """

    def __init__(self, extractor="IdentityExtractor", **args):
        self.featExtractor = util.lookup(extractor, globals())()
        PacmanQAgent.__init__(self, **args)
        self.weights = util.Counter()

    def getWeights(self):
        return self.weights

    def getQValue(self, state, action):
        """
        Should return Q(state,action) = w * featureVector
        where * is the dotProduct operator
        """

        Q_value = 0
        for feature in self.featExtractor.getFeatures(state, action):
            Q_value += self.weights[feature] *
self.featExtractor.getFeatures(state, action)[feature]
        return Q_value

    def update(self, state, action, nextState, reward):
        """
        Should update your weights based on transition
        """

        max_next_Q_value = -sys.maxsize
        for legal_action in self.getLegalActions(nextState):
            if self.getQValue(nextState, legal_action) > max_next_Q_value:
                max_next_Q_value = self.getQValue(nextState, legal_action)
        if max_next_Q_value == -sys.maxsize:
            max_next_Q_value = 0
        difference = (reward + (self.discount * max_next_Q_value)) -
self.getQValue(state, action)
        self.qvalue[(state, action)] += self.alpha * difference
        for feature in self.featExtractor.getFeatures(state, action):
            self.weights[feature] += self.alpha * difference *
self.featExtractor.getFeatures(state, action)[feature]

    def final(self, state):
        "Called at the end of each game."
        # call the super-class final method
```

```
        PacmanQAgent.final(self, state)

        # did we finish training?
        if self.episodesSoFar == self.numTraining:
            # you might want to print your weights here for debugging
            "*** YOUR CODE HERE ***"
            pass
```

در پیاده‌سازی این عامل، هدف تعیین وزن‌هایی‌ست که بتوانند پس از تجربه تعدادی sample، ارزش‌هایی برای حالت‌های مختلف مسئله تعیین کنند. برای این کار برای هر حالت تعدادی فیچر تعریف می‌شوند که برای همه حالت‌ها تعداد و نوع آن‌ها یکسان است. سپس وزن هر فیچر تعیین شده و با رخ دادن تجربه‌های جدید این وزن‌ها بروزرسانی می‌شوند.

$$Q(s, a) = \Sigma(n)(i=1)\ f_i(s, a) * w_i$$

مقادیر Q value برای هر حالت از رابطه بالا محاسبه می‌شوند.

سپس برای بروزرسانی مقادیر وزن‌ها از روابط زیر استفاده می‌کنیم:

$$w_i \leftarrow w_i + \alpha * \text{difference} * f_i(s, a)$$

$$\text{difference} = (r + \gamma \max_{a'} Q(s', a')) - Q(s, a)$$

در رابطه با نحوه پیاده‌سازی، در تابع getQValue مانند اولین رابطه ذکر شده، مقادیر Q value هر حالت بر اساس فیچرها و وزن‌های آن‌ها محاسبه و بازگردانی می‌شود.

همچنین در تابع update، وزن هر فیچر بر اساس حاصل تفاضل مقدار واقعی حالت و مقدار محاسبه شده توسط وزن‌ها و با در نظر گرفتن ضریب یادگیری، بروزرسانی می‌شود.