۱.

پیاده سازی معادله ی تکرار ارزش

برای این کار یک حلقه به تعداد تکرار ها می نویسیم (در هر تکرار یک Counter می سازیم از کلاس util برای نگهداری مقدار V های جدید استیت ها و آن را در آخر هر تکرار جاگذاری می کنیم در مقادیر valueی استیت هایمان)

یک حلقه بر روی state ها می نویسیم

یک حلقه روی اکشن های هر استیت می نویسیم. اکشن های ممکن برای هر استیت را از تابع getPossibleActions می گیریم (از اینجا به بعد، بعد از سیگما است)

یک حلقه روی احتمال های هر اکشن می نویسیم و نهایتا مقدار value های ممکن را برای هر استیت محاسبه می کنیم و ماکسیمم آن ها را ذخیره می کنیم. آخر هر تکرار مقدار value های استیت هایمان را آپدیت می کنیم (تابع getTransitionStatesAndProbs دوتایی های شامل استیت بعدی و احتمالی که با آن اکشن به آن استیت بعد می رویم را بر می گرداند)

به این صورت این معادله پیاده سازی می شود. (می توانستیم از تابع computeQValueFromValues هم استفاده کنیم اما کل معادله ی بلمن را برای کامل تر بودن همینجا نوشته ایم)

```
def runValueIteration(self):
    # Write value iteration code here
    "*** YOUR CODE HERE ***"
    for i in range(self.iterations):
        tempValues = util.Counter()

        for state in self.mdp.getStates():
            max = math.inf*-1
            tempValues[state] = 0.0

            for action in self.mdp.getPossibleActions(state):
                tempValue = 0.0
                t = self.mdp.getTransitionStatesAndProbs(state, action)

                for (nextS, p) in t:
                    r = self.mdp.getReward(state, action,nextS)
                    nextSValue = self.values[nextS]
                    tempValue += p*(r + self.discount * nextSValue)

                if tempValue > max:
                    max = tempValue

                tempValues[state] = max

        self.values = tempValues
```

پیاده سازی متد بعدی

برای محاسبه ی q-value یک استیت با اکشن مورد نظر از روی value ها باید یک حلقه بر روی احتمال های آن اکشن بنویسیم و با در نظر گرفتن reward و value های استیت هایمان و احتمال رفتن به هر استیت بعدی، معادله زیر را پیاده سازی کنیم و q-value را محاسبه کنیم (در قسمت قبل هم توضیح داده شد)

$$Q'(s,a) = \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma V'(s') \right]$$

```
def computeQValueFromValues(self, state, action):
    """
      Compute the Q-value of action in state from the
      value function stored in self.values.
    """
    "*** YOUR CODE HERE ***"
    tempValue = 0
    t = self.mdp.getTransitionStatesAndProbs(state, action)
    for (nextS, p) in t:
        r = self.mdp.getReward(state, action, nextS)
        nextSValue = self.getValue(nextS)
        tempValue += p * (r + self.discount * nextSValue)

    return tempValue
```

متد بعدی محاسبه ی بهترین اکشن در استیت مورد نظر با کمک value های اتیست هاست

برای این کار یک حلقه روی اکشن های ممکن در آن استیت می نویسیم و مقادیر q-value را می گیریم و در بین اکشن ها آنی که بیشترین q-value را به ما می دهد بر می گردانیم

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^*(s')]$$

$$\pi^*(s) = \arg\max_a Q^*(s,a)$$

```
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 ***"
    if self.mdp.isTerminal(state):
        self.values[state] = 0
        return None
    optAction = None
    maxQ = math.inf*-1
    for action in self.mdp.getPossibleActions(state):
        qValue = self.computeQValueFromValues(state, action)
        if qValue >= maxQ:
            optAction = action
            maxQ = max(maxQ, qValue)
    return optAction
```

.۲

برای این که سیاست بهینه باعث شود که عامل به انتهای پل برسد، باید نویز را کم کنیم زیرا هر چه نویز کم تر باشد یعنی احتمال اینکه عامل ما همان اکشن داده شده را اجرا کند بیشتر می شود. نویز را از ۰/۲ کم می کنیم و زمانی جواب می گیریم که نویز را ۰/۰۱ قرار دادیم

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0.01
    return answerDiscount, answerNoise
```

.۳

برای ترجیح خرجی نزدیک تر و ریسک کردن صخره، باید مقدار living reward رو منفی تر کنیم تا زمانی که مسیر کوتاه تر را ترجیح دهد برای رسیدن به خروجی . زیرا با هر حرکتش مقدار زیادی reward را از دست می دهد (وقتی این مقدار را برابر -۳ گذاشتیم به جواب رسیدیم)

```python
def question3a():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = -3
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

برای ترجیح خروجی نزدیک تر و اجتناب از صخره ها باید مقدار living reward رو از قسمت a بیشتر کنیم تا مسیر طولانی تر رو ترجیح دهد (یعنی اجتناب کردن از صخره ها) و همچنین مقدار discount را کم تر می کنیم تا خروجی نزدیک تر را ترجیح دهد (در این صورت خروجی دورتر مقدار زیادی به خاطر discount از دست می دهد و ارزش خروجی نزدیک تر بیشتر می شود برایمان)

```python
def question3b():
    answerDiscount = 0.5
    answerNoise = 0.1
    answerLivingReward = -1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

برای ترجیح خروجی دور تر و اجتناب از صخره ها تنها تفاوتی در مقادیر قسمت b می دهیم این است که مقدار discount را بیشتر می کنیم تا خروجی دورتر را ترجیح دهد (در این صورت چون discount زیاد است، مقدار خروجی دور تر با تخفیف بیشتر همچنان از خروجی نزدیک تر با تخفیف کمتر بیشتر است)

```python
def question3c():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = -1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

برای این قسمت تنها تفاوتی که با قسمت c می دهیم این است که living reward را بیشتر می کنیم تا ریسک نکردن و مسیر طولانی تر به ریسک کردن بیارزد (زیرا در هر مرحله reward زیادی از دست می دهد پس می خواهد تعداد گام ها کم تر باشد)

```python
def question3d():
    answerDiscount = 0.9
    answerNoise = 0.1
    answerLivingReward = -0.1
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

برای این قسمت فقط کافیست living reward را تغییر دهیم و مقدار آن را مثبت کنیم تا در حلقه بیفتیم و فقط بخواهیم living reward جمع کنیم

```python
def question3e():
    answerDiscount = 1
    answerNoise = 0.1
    answerLivingReward = 0.0001
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

.٤

برای این بخش یک حلقه روی تعداد تکرار می نویسیم. از آنجایی که در هر تکرار فقط می خواهیم یک مقدار value استیت را تغییر دهیم از indexing و i استفاده می کنیم و روی اکشن های ممکن استیت (i % len(states ام یک حلقه می نویسیم و همچنین یک حلقه روی احتمالات هر اکشن می نویسیم مقادیر ممکن برای Vk+۱ را محاسبه و ماکسیمم مقدار را برای value ی آن استیت مورد نظر قرار می دهیم

```python
def runValueIteration(self):
    "*** YOUR CODE HERE ***"
    for i in range(self.iterations):
        state = self.mdp.getStates()[i % len(self.mdp.getStates())]
        if self.mdp.isTerminal(state):
            continue
        max = -1 * math.inf
        for action in self.mdp.getPossibleActions(state):
            tempValue = 0.0
            t = self.mdp.getTransitionStatesAndProbs(state, action)
            for (nextS, p) in t:
                reward = self.mdp.getReward(state, action, nextS)
                nextSvalue = self.values[nextS]
                tempValue += p*(reward + self.discount*nextSvalue)
            if tempValue > max:
                max = tempValue
        self.values[state] = max
```

.٥

مراحل پیاده سازی این بخش دقیقا به همان ترتیب گفته شده در دستورکار است :)

- ابتدا باید برای هر حالت، همه پسینها مشخص شود.
- یک صف خالی برای نگهداری اولویتها تعریف کنید.
- برای هر حالت غیرپایانی s:

- قدر مطلق تفاضل بین مقدار فعلی حالت s(که در values.self نگهداری میشود(و بیشترین مقدار Q ممکن از حالت s که با استفاده از اقدام های ممکن قابل تعریف است را محاسبه کنید. این مقدار را diff بنامید.
- حالت s را با اولویت diff -به صف اولویتها اضافه کنید. (دلیل اینکه از اولویت منفی استفاده میکنیم این است که صف اولویتها به صورت heap min است و اولویت با مقدار عددی کمتر به معنی ارجحیت است و ما میخواهیم حالتی که بیشترین خطا را دارد اولویت بیشتری داشته باشد و زودتر به روزرسانی شود)

```
    def runValueIteration(self):
        "*** YOUR CODE HERE ***"
        tempValues = self.values
        priorityQueue = util.PriorityQueue()
        predecessors = {}
        for state in self.mdp.getStates():
            predecessors[state] = set()

#FOR 1
        for state in self.mdp.getStates():
            qvalues = util.Counter()
            currentValue = tempValues[state]                    #value for the current state

            for action in self.mdp.getPossibleActions(state):
                t = self.mdp.getTransitionStatesAndProbs(state, action)

                for (nextS, p) in t:
                    if p != 0.0:                                #p > 0, add the state to the predecessors
                        predecessors[nextS].add(state)

                qvalues[action] = self.computeQValueFromValues(state, action)
            diff = abs(currentValue - qvalues[qvalues.argMax()])   #diff = current value - max QValue
            priorityQueue.update(state, -diff)                  #Insert that state into priority queue
```

- به ازای تعداد تکرارهای مشخص شده (self.iterations) :
  - اگر صف اولویتها خالی میباشد کار پایان یافته است.
  - در غیر این صورت حالت s را از صف اولویتها بردارید.
  - در صورتی که s حالت پایانی نبود، مقدار حالت s را)در values.self)بهروز رسانی کنید.
  - به ازای هر پسین p از حالت s:
    - قدر مطلق تفاضل بین مقدار فعلی حالت p(که در values.self نگهداری میشود(و بیشترین مقدار Q ممکن از حالت p که با استفاده از اقدام های ممکن قابل تعریف است را محاسبه کنید. این مقدار را diff بنامید.
    - اگر theta < diff بود حالت p را با اولویت diff -به صف اولویتها اضافه کنید. البته فقط در صورتی که حالت p با اولویت مساوی یا کمتر در صف وجود نداشته باشد.

```
#FOR 2
        for i in range(self.iterations):
            if priorityQueue.isEmpty():                    #if the priority queue is not empty, continue
                return
            state = priorityQueue.pop()                    #Pop a state from the queue

            if not self.mdp.isTerminal(state):             #calculate the best Q-value if not a terminal state
                QValues = util.Counter()
                for action in self.mdp.getPossibleActions(state):
                    QValues[action] = self.computeQValueFromValues(state, action)

                tempValues[state] = QValues[QValues.argMax()]

            for p in predecessors[state]:
                QValues_p = util.Counter()
                for action in self.mdp.getPossibleActions(p):    #find the best action with the highest QValue
                    QValues_p[action] = self.computeQValueFromValues(p, action)

                diff = abs(tempValues[p] - QValues_p[QValues_p.argMax()])   #diff = current value - best possible value
                if (diff > self.theta):
                    priorityQueue.update(p, -diff)
```

.٦

ابتدا در constructor ، qValues را از جنس Counter می سازیم برای نگهداری مقدار های Q برای استیت ها و اکشن های ممکنشان (از آنجایی که از Counter استفاده کردیم مقداردهی اولیه هم صفر است)

برای گرفتن یک Q-value ی مخصوص، کافیست که استیت و اکشن مورد نظرمان را به متد getQValue بدهیم و این متد هم مقدار Qی مورد نظر را از همان دیکشنری qValues که در قسمت قبل گفتیم می گیریم (کلید آن همان دوتایی state و action است و مقدار کلید همان مقدار Qی مورد نظر است)

برای محاسبه ی valueی یک استیت توسط Q-valueهایش کافیست معادله ی زیر را پیاده سازی کنیم

برای این کار یک حلقه روی اکشن های ممکن این استیت می نویسیم و توسط متد getQValue ، Q-value مربوط به این اکشن ها را برای استیتمان می گیریم و بین آن ها ماکسیممشان را برای value ی استیتمان قرار می دهیم

$$V^*(s) = \max_a Q^*(s, a)$$

```
def __init__(self, **args):
    "You can initialize Q-values here..."
    ReinforcementAgent.__init__(self, **args)
    "*** YOUR CODE HERE ***"
    self.qValues = 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
    """
    "*** YOUR CODE HERE ***"
    return self.qValues[(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.
    """
    "*** YOUR CODE HERE ***"
    tmpQvalues = util.Counter()
    for action in self.getLegalActions(state):
        tmpQvalues[action] = self.getQValue(state, action)

    return tmpQvalues[tmpQvalues.argMax()]
```

این متد مانند قسمت قبلی است یعنی بیشترین Q-value را پیدا می کنیم و اکشنی که این Q-value را به ما می دهد را بر می گردانیم (توسط argmax)

```
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.
    """
    "*** YOUR CODE HERE ***"
    tmpQvalues = util.Counter()
    for action in self.getLegalActions(state):
        tmpQvalues[action] = self.getQValue(state, action)

    return tmpQvalues.argMax()
```

برای آپدیت کردن Q-valueی مربوط به اکشن داده شده ی استیت مان، باید فرمول های زیر را پیاده سازی کنیم

برای این کار توسط همان متد getQValue مقدار Q-value استیت و اکشن داده شده را می گیریم. متغیر sample را به همین صورت فرمول داده شده مقدار می دهیم و با آن و آلفایی که داریم مقدار Q-value را به صورت زیر آپدیت می کنیم

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s') \left[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')\right]$$

$$sample = R(s,a,s') + \gamma \max_{a'} Q(s',a')$$

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

```
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 ***"
    Q = self.getQValue(state, action)
    sample = reward + self.discount * self.computeValueFromQValues(nextState)     #sample = r + discount * maxQValue
    self.qValues[(state, action)] = (1 - self.alpha) * Q + (self.alpha) * sample  #Q = (1 - alpha) * Q + alpha * sample
```

.۷

برای پیاده سازی این بخش با راهنمایی داخل دستورکار از تابع flipCoin(epsilon) از کلاس util استفاده می کنیم. این تابع یک عدد رندوم بین ۰ و ۱ انتخاب می کند و با اپسیلون داده شده مقایسه می کند، اگر کوچک تر بود باید رندوم عمل کند (توسط) و اگر بزرگ تر بود باید با بهترین سیاست عمل می کنیم

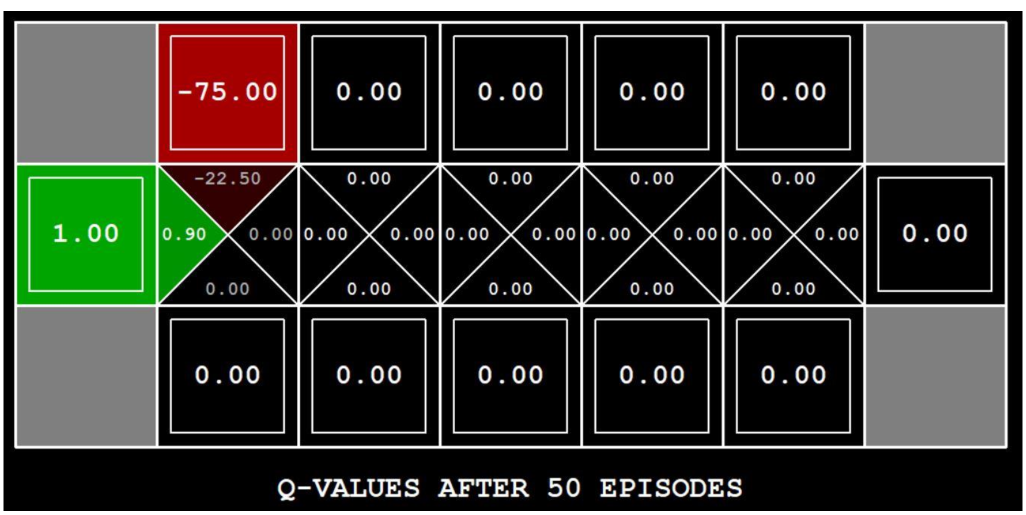
همچنین طبق چیزی که در کامنت این متد نوشته شده باید قبل از این کار چک کنیم که اگر اکشن مجاز نداریم None را برگردانیم

```python
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 ***"
    if len(legalActions) == 0:
        return action
    if (util.flipCoin(self.epsilon)):
        action = random.choice(legalActions)           #random action
    else:
        action = self.computeActionFromQValues(state)      #optimal action

    return action
```

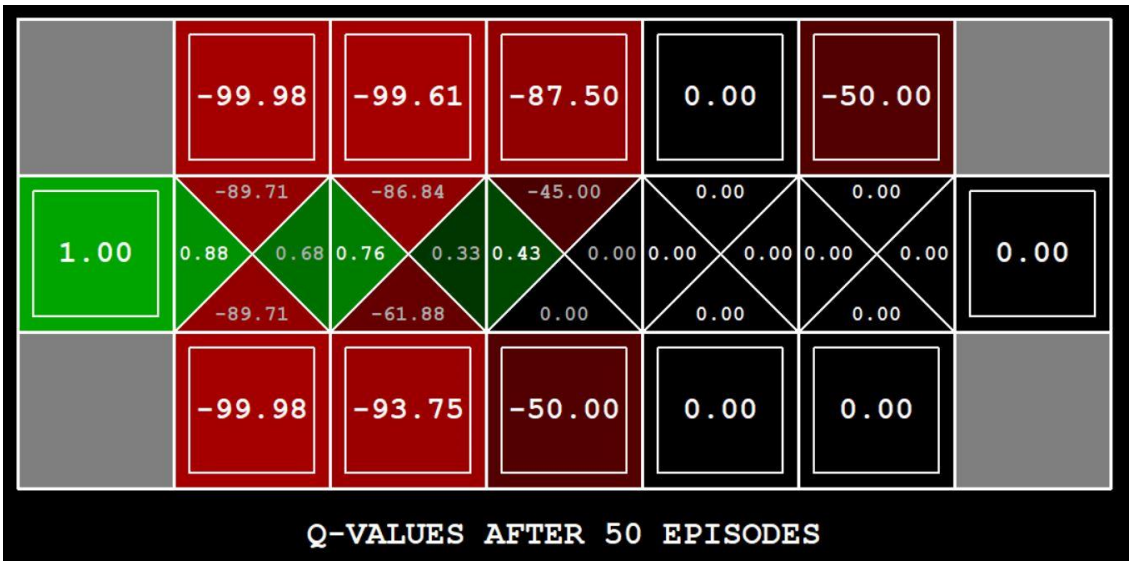
.۸

بررسی کنید:

با اپسیلون ۰ نتوانستیم سیاست بهینه را پیدا کنیم. وقتی اپسیلون صفر است یعنی رندوم عمل نمی کنیم و فقط exploit می کنیم در نتیجه فقط مقادیر یک استیت را آپدیت می کند



می بینیم که حتی با اپسیلون ۱ هم نتوانستیم در ۵۰ اپیزود سیاست بهینه را پیدا کنیم. هنگامی که اپسیلون ۱ است یعنی تماما رندوم عمل میکنیم و در حال کاوش هستیم و خانه های بیشتری را کاوش می کنیم. در نتیجه اپسیلون هر چه که باشد بعید است که به خروجی دورتر برسد



با تغییر نرخ یادگیری هم نمیتوانیم به خروجی دور تر برسیم

حال با چک کردن مقادیر مختلف نرخ یادگیری و اپسیلون متوجه می شویم که نمیتوانیم مقادیری به آن ها نسبت بدهیم که در ۵۰ اپیزود به سیاست بهینه دست پیدا کنیم

۹.

می بینیم که عامل ما تمام بازی ها را برده است در نتیجه از اطلاعات کسب شده به درستی سیاست بهینه را محاسبه کرده است

```
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 503
Pacman emerges victorious! Score: 495
Pacman emerges victorious! Score: 499
Pacman emerges victorious! Score: 499
Average Score: 499.4
Scores:        503.0, 495.0, 503.0, 499.0, 495.0, 503.0, 503.0, 495.0, 499.0, 499.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

۱۰.

برای پیاده سازی متد getQValue مقادیر fها و wها را می گیریم و با یک حلقه روی آن ها فرمول زیر را پیاده می کنیم

$$Q(s,a) = \sum_{i=1}^{n} f_i(s,a) w_i$$

```
def getQValue(self, state, action):
    """
      Should return Q(state,action) = w * featureVector
      where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    fs = self.featExtractor.getFeatures(state, action)
    ws = self.getWeights()
    q = 0
    for f in fs:
        q += fs[f] * ws[f]      #Q = sigma (feature i * weight i)

    return q

def update(self, state, action, nextState, reward):
    """
       Should update your weights based on transition
    """
    "*** YOUR CODE HERE ***"
    fs = self.featExtractor.getFeatures(state, action)

    #difference = (r + discount * maxQ(nextState, a')) - Q(state, action)
    difference = (reward + (self.discount * self.getValue(nextState))) - self.getQValue(state, action)

    for f in fs:      # w i = w i +  a * difference * f i
        self.weights[f] = self.weights[f] + (self.alpha * difference * fs[f])
```

برای پیاده سازی متد update هم باید مقدار difference را با فرمول داده شده در زیر محاسبه کنیم

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

و سپس یک حلقه روی f ها می زنیم و وزن هایشان را با فرمول زیر آپدیت می کنیم

$$w_i \leftarrow w_i + \alpha \cdot difference \cdot f_i(s,a)$$