

# به نام خدا

گزارشکار پروژه سوم هوش مصنوعی

هلیاسادات هاشمی پور - ۹۸۳۱۱۰۶

## بخش اول، تکرار ارزش

در این بخش با توجه به خواسته دستوکار عمل می کنیم. در اصل، عامل تکرار ما به شکل آفلاین می باشد و به صورت reinforcement agent نمی باشد. همانطور که در دستورکار هم ذکر شده است تکرار ارزش، تخمین های  $k$  مرحله ای مقادیر بهینه  $V_k$  را حساب می کنیم. معادله زیر را می خواهیم پیاده سازی کنیم.

$$V_{k+1} \leftarrow \max_a \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V_k(s')]$$

بنابراین سهتابع برای پیاده سازی معادله بالا در نظر می گیریم:

- تابع `runValueIteration` که در واقع تابع اصلی است و تکرار ارزش در این تابع صورت می گیرد و از توابع پیاده سازی شده استفاده می شود. در این بخش از ورژن batch تکرار ارزش، استفاده می کنیم. در واقع `value` های استیت ها در تکرار قبلی برای به روزرسانی مقادیر استیت ها در تکرار بعدی استفاده می شود. با توجه به کد، توابع استفاده شده در کد به صورت زیر عمل میکنند: از تابع `getState()` همه استیت های در محیط را به ما برمیگرداند. تابع `getPossibleActions(state)` همه `action` های `action` که مورد قبول است را به ما می دهد. با توجه به کد، ما ابتدا مقداردهی اولیه می کنیم، سپس در `while` از توابع توضیح داده شده استفاده می کنیم. اگر استیت ما `terminal` نباشد باید بهترین مقدار را به عنوان حداقل مجموع مورد انتظار پاداش اکشن های مختلف پیدا کنیم. `util.Counter` مقادیر برای `action` های استیت فعلی را می دهد. در هر تکرار که صورت می گیرد. دو لوب تو در تو داریم که در هر تکرار بزرگترین مقدار را در `maxValue` می ریزیم و در متغیر `newValue` در استیت مورد نظر آن را میریزیم. در آخر مقدار جدید(`newValue`) را کپی می کنیم.)

```

def runValueIteration(self):
    # Write value iteration code here
    """ YOUR CODE HERE """
    i = 0
    states = self.mdp.getStates()
    iteration = self.iterations

    for state in states:
        self.values[state] = 0.0

    while i < iteration:
        newValue = util.Counter()
        for state in states:
            if self.mdp.isTerminal(state):
                continue

            maxVal = float("-inf")
            actions = self.mdp.getPossibleActions(state)

            for action in actions:
                QValue = self.computeQValueFromValues(state, action)
                maxVal = max(maxVal, QValue)
                # update newValue at state s to largest QValue
                newValue[state] = maxVal
        self.values = newValue.copy()
        i += 1

```

- تابع `computeQValueFromValues` طبق آنچه که در دستور کار گفته شده است، اکشن در استیت را با توجه به تابع مقدار در `self.values` محاسبه می کند. ابتدا `Q-value` مقدار متغیر را به صفر مقداردهی اولیه میکنیم. حال `transition` ( $QValue=0$ ) حالت استیت ها `.getTransitionStatesAndProbs` و `probability` را حساب می کنیم با استفاده از دستور `value` ، `transition` به عنوان جمع پاداش برای رسیدن به آن `transition` و `discount` مقدار `Qvalue` را می دهد. که آن را به عنوان خروجی برمی گردانیم.

همان گاما ما می باشد.

```

def computeQValueFromValues(self, state, action):
    """
    Compute the Q-value of action in state from the
    value function stored in self.values.
    """
    value = 0.0
    transitions = self.mdp.getTransitionStatesAndProbs(state, action)

    for transition in transitions:
        transitionReward = self.mdp.getReward(state, action, transition[0])
        value += transition[1] * (transitionReward
                                + self.discount * self.values[transition[0]])

    QValue = value
    return QValue

```

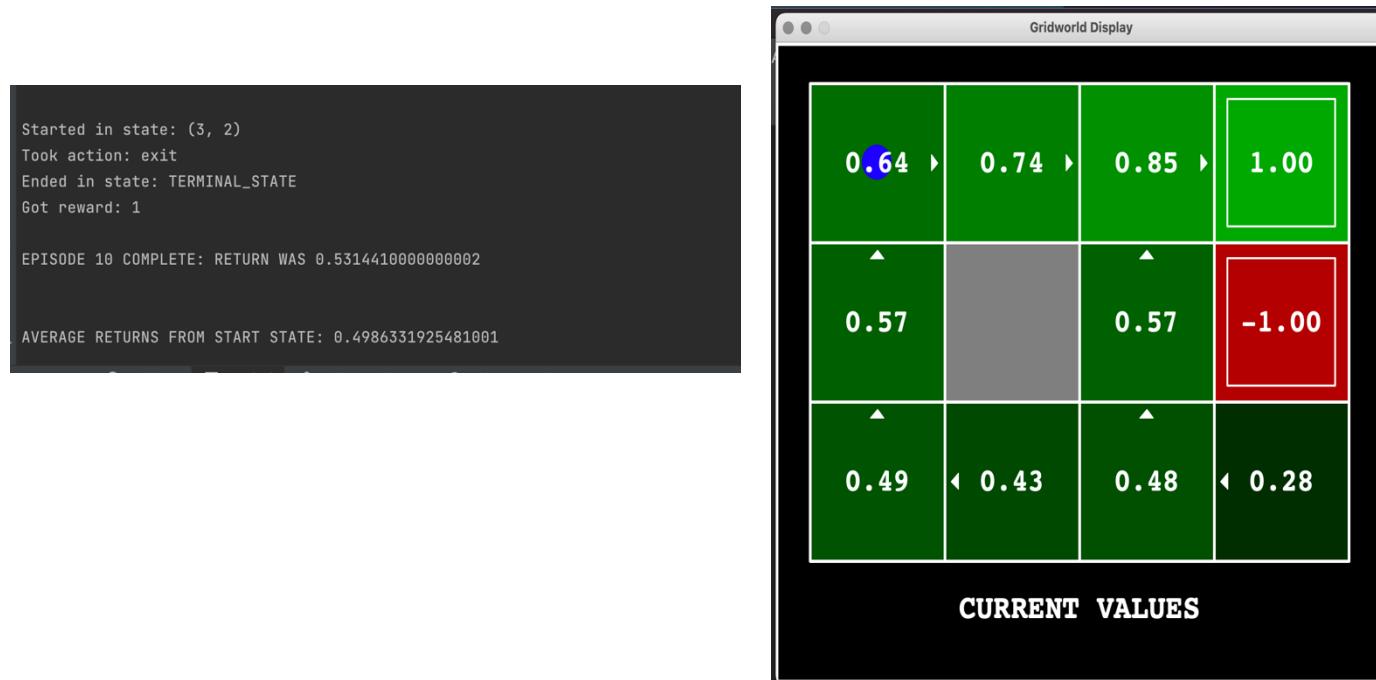
تابع `computeActionFromValues` داده value function، بهترین عمل را با توجه به شده توسط `self.values` محاسبه می کند. در اصل `policy` که بهترین اکشن در استیت داده شده است. در لوب تابع `computeQValueFromValues` را فراخوانی کرده، سپس ماکسیمم مقدار را با توجه به شرط نوشته شده در هر تکرار پیدا کرده و `action` مربوطه را در قرار می دهیم و آن را به عنوان خروجی تابع بر می گردانیم.

```
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.
    """
    actions = self.mdp.getPossibleActions(state)
    maxVal = float('-inf')
    maxAction = 0.0

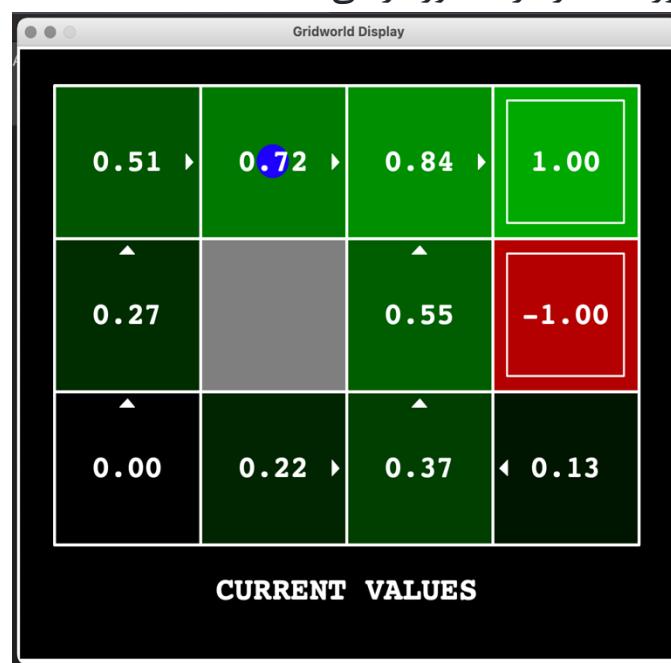
    for action in actions:
        QValue = self.computeQValueFromValues(state, action)
        if QValue > maxVal:
            maxVal = QValue
            maxAction = action

    return maxAction
```

دستور `python gridworld.py -a value -i 100 -k 10` را در ترمینال می زنیم و نتایج زیر حاصل می شود.

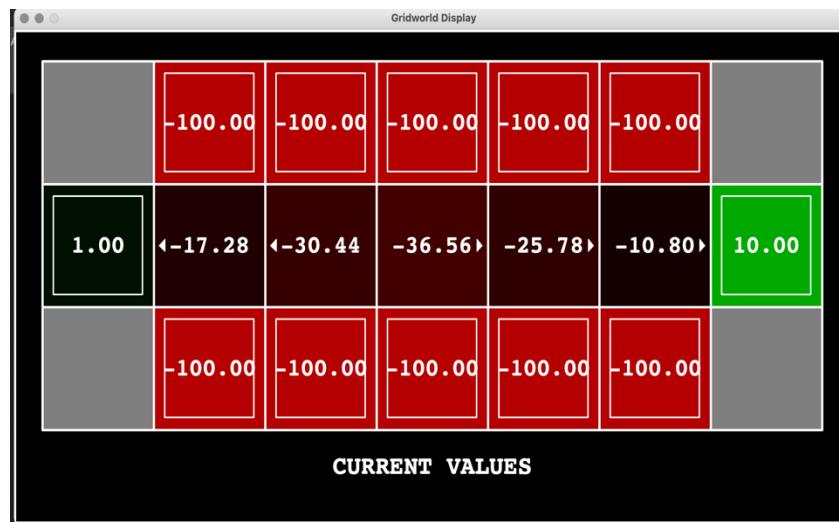


حال دستور `python gridworld.py -a value -i 5` را می زنیم و خروجی زیر حاصل می شود.(همان خروجی مورد انتظار در دستور کار می باشد)



## بخش دوم، تجزیه و تحلیل عبور از پل

مقدار پیش فرضی که برای تخفیف و نویز در نظر گرفته شده به ترتیب  $0.2 \cdot 0.9$  می باشد.  
خروجی با مقادیر پیش فرض به صورت زیر است:



EPISODE 1 COMPLETE: RETURN WAS -90.0

AVERAGE RETURNS FROM START STATE: -90.0

حال ما باید یکی از آن ها را تغییر دهیم تا سیاست بهینه باعث شود عامل ما برای عبور از پل تلاش کند. من در این بخش `answerNoise` را به صفر تغییر دادم. در اصل با نویز صفر (تقریباً صفر در نظر گرفتم) عامل ما برای عبور از پل تلاش می کند. (با نویز صفر در اصل مسیله ما قطعی

می شود)

```
def question2():
    answerDiscount = 0.9
    answerNoise = 0.000000000001
    return answerDiscount, answerNoise
```

حال خروجی ما به شکل زیر است:

EPISODE 1 COMPLETE: RETURN WAS 0.9

AVERAGE RETURNS FROM START STATE: 0.9

## بخش سوم، سیاست ها

در نقشه DiscountGrid، سیاست بهینه های خواسته شده بر اساس تخفیف، نویز و پاداش زندگی ایجاد می شود.

در بخش 3a، چون عامل ما باید به سرعت خارج شود (در اصل خروجی نزدیک را ترجیح می دهد که پاداش آن  $+1$  است) پاداش منفی نگه داشته می شود تا او سعی کند که به سرعت خارج شود. (عامل ما خطر صخره را هم ریسک میکند. پس نویز را صفر در نظر گرفته ام.)

```
def question3a():
    answerDiscount = 0.1
    answerNoise = 0
    answerLivingReward = -5
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

بخش 3b عامل خروجی نزدیک را ترجیح می دهد( $+1$ ) اما از صخره اجتناب می کند(پس نویز را اضافه میکنیم) و راه طولانی تر را انتخاب میکند.

```
def question3b():
    answerDiscount = 0.1
    answerNoise = 0.1
    answerLivingReward = -2
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

بخش 3c عامل ما خروجی دور را ترجیح می دهد( $+10$ ) و خطر، ریسک صخره را هم داریم. (پس نویز را صفر در نظر گرفته ام) بنابراین پاداش به دست آمده از قسمت 3a بیشتر می باشد.

```
def question3c():
    answerDiscount = 1
    answerNoise = 0
    answerLivingReward = -2
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

بخش 3d عامل ما خروجی دور را ترجیح می دهد(+) تخفیف را ۱ در نظر می گیریم. از خطر صخره اجتناب می کند.(پس نویز را در نظر گرفته ام) بنابراین پاداش ما از بخش 3c بیشتر می شود.(در اصل کم تر منفی است)

```
def question3d():
    answerDiscount = 1
    answerNoise = 0.1
    answerLivingReward = -0.4
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

بخش 3e عامل ما باید از خروجی و صخره اجتناب کند من استدلالی که کردم این بود که مقدار پاداش را زیاد در نظر بگیرم که عامل ما خارج نشود و اینکه برای اینکه از صخره اجتناب کند نویز اضافه کردم.

```
def question3e():
    answerDiscount = 1
    answerNoise = 0.1
    answerLivingReward = 1000
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

## بخش چهارم، تکرار ارزش ناهمزمان

در این بخش، در هر تکرار ما فقط یک حالت را به روزرسانی می کنیم. به این صورت که در اولین تکرار ما فقط مقدار حالت اول را در لیست حالت ها آپدیت می کنیم و این روند ادامه دارد تا جایی که مقدار هر حالت یکبار به روز شوند. سپس از حالت اول برای تکرار بعدی استفاده می شود. ارث بری می کند پس من یک متده به اسم `runValueIteration` پیاده سازی کرده ام. بنابراین کد خواسته شده طبق خواسته سوال می باشد. در هر تکرار، مقدار فقط یک استیت آپدیت می شود (در اصل چرخش در لیست استیت ها صورت می گیرد) پس فرق این بخش و بخش نخست همین می باشد. حال برای استیت ما دیگر یک لوپ جدا نداریم و کاری که می کنیم این است که مُد ایندکس در نظر گرفته شده را، بر طول استیت های در محیط، در نظر می گیریم. در لوپ بعدی هم ماسکسیمم مقدار را پیدا کرده (مثل قسمت های قبل) سپس بعد از انجام تکرار `self.values[state]` را با مقدار به دست آمده مقداردهی می کنیم. بنابراین در هر تکرار یک حالت به روزرسانی می شود و در تکرار بعدی حالت دیگر را آپدیت می کنیم.

```
def runValueIteration(self):
    """ YOUR CODE HERE """
    i = 0
    states = self.mdp.getStates()
    iteration = self.iterations

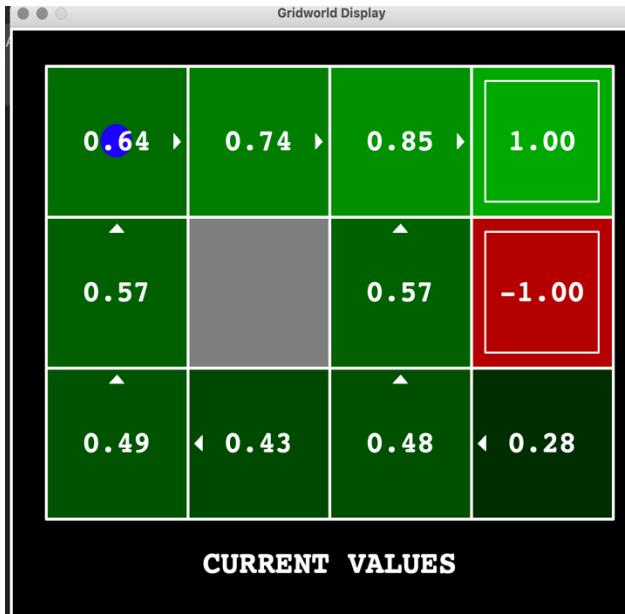
    for state in states:
        self.values[state] = 0.0

    while i < iteration:
        stateIndex = i % len(states)
        state = states[stateIndex]

        if not self.mdp.isTerminal(state):
            maxVal = float("-inf")
            actions = self.mdp.getPossibleActions(state)

            for action in actions:
                QValue = self.computeQValueFromValues(state, action)
                maxVal = max(maxVal, QValue)
            self.values[state] = maxVal
        i += 1
```

دستور `python gridworld.py -a asynchvalue -i 1000 -k 10` را در ترمینال می زنیم و نتایج زیر حاصل می شود.



```
EPISODE 10 COMPLETE: RETURN WAS 0.5314410000000002
AVERAGE RETURNS FROM START STATE: 0.5225246010000001
```

## بخش پنجم، تکرار ارزش اولویت بندی شده

در این بخش طبق مراحل خواسته شده در دستور کار عمل کردم، یعنی ابتدا برای همه حالت ها dict precedessors را مشخص می شود. (در کد به صورت precedessors معرفی کرده ام) بنابراین precedessors را بر اساس شرایط (در حلقه تکرار ما اگر احتمال، بیشتر از صفر باشد) موجود مقداردهی می کنیم.

سپس یک صف خالی برای نگهداری اولویت ها تعریف کردیم (priorityQueue). در حلقه تکرار در هر حالت non-terminal استیت s، کاری که انجام دادم این بود که بیشترین مقدار Q-value را پیدا می کنیم. لوب نوشته شده یک متغیر به اسم diff در نظر گرفتم که در هر تکرار مقدار قدرمطلق تفاضل میان بیشترین مقدار Q ممکن از استیت s و مقدار فعلی s را حساب می کند. حال در صف اولویت خود، s را push می کنیم (با اولویت (-diff)).

```
def runValueIteration(self):
    """*** YOUR CODE HERE ***"""
    mdp = self.mdp
    theta = self.theta
    states = mdp.getStates()
    predecessors = {} # dict
    priorityQueue = util.PriorityQueue() # Initialize an empty priority queue

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

    for s in states:
        if not mdp.isTerminal(s):
            actions = self.mdp.getPossibleActions(s)

            for action in actions:

                transitions = mdp.getTransitionStatesAndProbs(s, action)
                maxVal = float('-inf')

                for nextState, prob in transitions:
                    if prob > 0:
                        predecessors[nextState].add(s)

                QValue = self.computeQValueFromValues(s, action)
                maxVal = max(QValue, maxVal)
                diff = abs(maxVal - self.values[s])
                priorityQueue.push(s, - diff)
```

در حلقه تکرار بعدی، ابتدا چک می کنیم که اگر صف اولویت ما خالی باشد، break داده به معنای آنکه کار ما تمام شده است. اگر صف اولویت خالی نباشد استیت s را از صف اولویت می کنیم.

```
# now for the actual iterations
i = 0
iteration = self.iterations
while i < iteration:

    if priorityQueue.isEmpty():
        break
    else:
        s = priorityQueue.pop()
```

حال بررسی می کنیم که اگر استیت s، استیت non-terminal بود در شرط نوشته شده می خواهیم مقدار استیت s را در self.value آپدیت کردم. (کد این قسمت نوشته شده در قسمت های پیشین است در اصل در هر تکرار ماکسیمم مقدار QValue را پیدا کرده و در آخر self.values[s] را با مقدار به دست آمده یعنی maxVal آپدیت می کنیم)

```
else:
    s = priorityQueue.pop()
    if self.mdp.isTerminal(s):
        continue
    else:
        maxVal = float('-inf')
        actions = self.mdp.getPossibleActions(s)
        for action in actions:
            QValue = self.computeQValueFromValues(s, action)
            if QValue > maxVal:
                maxVal = QValue
        self.values[s] = maxVal
```

حال در یک لوب، که به این صورت است که به ازای هر  $s$  برای  $p$ ،  $predecessor$  (به این صورت نوشته شده است:  $for p \in predecessor[s]$ ) باز هم یک متغیر  $diff$  در نظر گرفته که قدر مطلق تفاضل مقدار فعلی  $p$  در  $self.value$  و بیشترین مقدار  $Q$ -value برای تمام اکشن‌های ممکن از  $p$  می‌باشد. سپس شرط  $diff > \theta$  را برقرار باشد،  $p$  را به صف اولویت با اولویت  $-diff$  push کنیم. (البته در اینجا از  $update$  استفاده کرده‌ام)

```

for p in predecessors[s]:
    if self.mdp.isTerminal(p):
        continue
    else:
        maxVal = float('-inf')
        actions = self.mdp.getPossibleActions(p)
        for action in actions:
            QValue = self.computeQValueFromValues(p, action)
            if QValue > maxVal:
                maxVal = QValue
        diff = abs(maxVal - self.values[p])

        if diff > theta:
            priorityQueue.update(p, -diff)

    i += 1

```

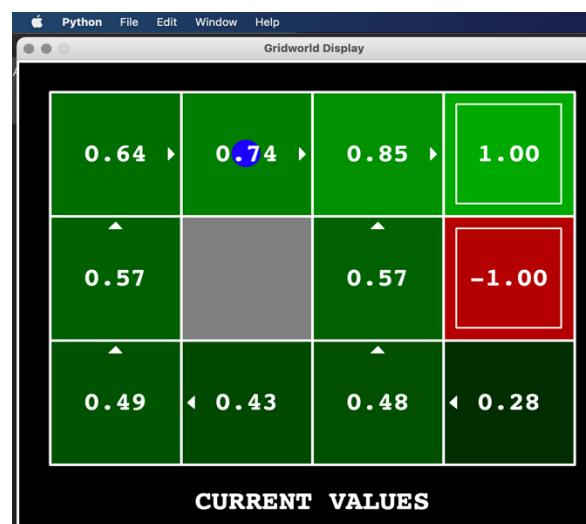
دستور `python gridworld.py -a priosweepvalue -i 1000` خروجی زیر را به ما می‌دهد.

```

EPISODE 1 COMPLETE: RETURN WAS 0.5904900000000002

AVERAGE RETURNS FROM START STATE: 0.5904900000000002

```



## بخش ششم، یادگیری Q

در این بخش از ما خواسته شده است که یک عامل یادگیری Q را بنویسیم که در اصل در هنگام ایجاد، تلاش زیادی برای یادگیری نمی کند. با استفاده از تابع update و آزمون و خطأ با تعامل با محیط، یاد میگیرد.

در این بخش متدهای زیر پیاده سازی شده اند:

- تابع `getValue`، این تابع اگر گره مقدار Q را نبیند باید مقدار 0.0 را بر گرداند، در غیر اینصورت آنچه که بر می گرداند  $Q(state,action)$  می باشد و همانطور که در کد هم قابل مشاهده می باشد در شروط پیاده سازی شده مقادیر خواسته شده بر گردانده می شود.

```
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 ***"
    QValue = self.qValues[(state, action)]
    if (state, action) not in self.qValues:
        return 0.0
    else:
        return QValue
```

- تابع `computeValueFromQValue` ای وجود نداشته باشد مقدار 0.0 برگردانده می شود. در این متده باید ماکسیمم  $Q(state, action)$ ،  $action$  را برگرداند. در حلقه تکراری که داریم، ابتدا مقدار  $QValue$  را با استفاده از `getQValue` گرفته سپس در شرطی که گذاشتیم ماکسیمم مقدار را پیدا می کنیم و در `max_action` میریزیم در آخر هم `max_action` را بر می گردانیم.

```

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 ***"
    actions = self.getLegalActions(state)
    max_action = float('-inf')

    if len(actions) == 0:
        return 0.0

    maxVal = float('-inf')
    for action in actions:
        QValue = self.getQValue(state, action)
        if QValue > maxVal:
            maxVal = QValue
            max_action = maxVal
    return max_action

```

- تابع `computeActionFromQValues` ای نداشته باشم باید `None` بر گردانده شود. این متده در اصل `bestaction` را برای رسیدن به استیت، محاسبه کنیم. برای این کار، ابتدا مقدار `maxValue` را مقداردهی کرده سپس در حلقه `for`، متغیر `value` را با استفاده از `getQValue` مقدار داده سپس ماکسیمم مقدار را در شرط `if` نوشته شده پیدا می کنیم. حال در آخر هم با توجه به توضیحات دستور کار برای اینکه متده نوشته شده بهتر عمل کند پیوند ها باید به صورت تصادفی قطع شوند. بنابراین با استفاده از `random.choice` و دادن `bestAction` به عنوان پارامتر ورودی به آن، مقدار تصادفی به دست آمده را بر می گردانیم.

```

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 ***"
    bestAction = None
    actions = self.getLegalActions(state)
    if len(actions) == 0:
        return None

    maxValue = float('-inf')

    for action in actions:
        value = self.getQValue(state, action)
        if value > maxValue:
            maxValue = value
            bestAction = [action]

    return random.choice(bestAction)

```

- تابع update، کد این قسمت با استفاده از روابط زیر نوشته شده است:

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

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

همانطور که می بینیم در خط نخست sample را محاسبه کرده سپس در خط بعدی هم

همانطور که می بینیم در خط نخست sample را محاسبه می کنیم.

```
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 ***"
    sample = reward + self.discount * self.getValue(nextState)
    self.qValues[(state, action)] = (1 - self.alpha) * self.qValues[(state, action)] + self.alpha * sample
```

## بخش هفتم، epsilon حریصانه

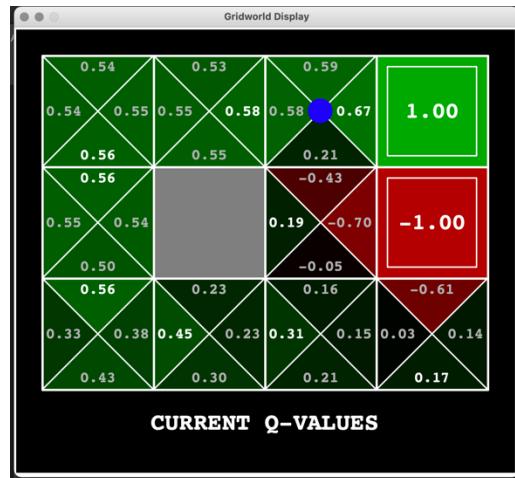
در این بخش تابع `getAction` را پیاده سازی کردم که در این متدها، اگر `legal action` ای نداشته باشیم باید `None` برگردانده شود. در غیر اینصورت، با توجه به احتمال `self.epsilon` (با توجه به شرط نوشته شده یعنی `util.flipCoin(p)` می‌توان تعیین کرد که به این شکل عمل می‌کند که را با احتمال `p` بر می‌گرداند، در غیر اینصورت `false` را با احتمال `1-p` بر می‌گرداند) `action` به شکل تصادفی (با استفاده از `random.choice(actions)`) انتخاب شود و در غیر اینصورت از بهترین مقادیر  $Q$  فعلی خود انتخاب می‌شود (`best policy action`). این کار در کد به صورت `self.getPolicy(state)` صورت می‌گیرد) در آخر هم به عنوان خروجی، مقدار `action` برگردانده می‌شود.

```
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)
    """
    actions = self.getLegalActions(state)

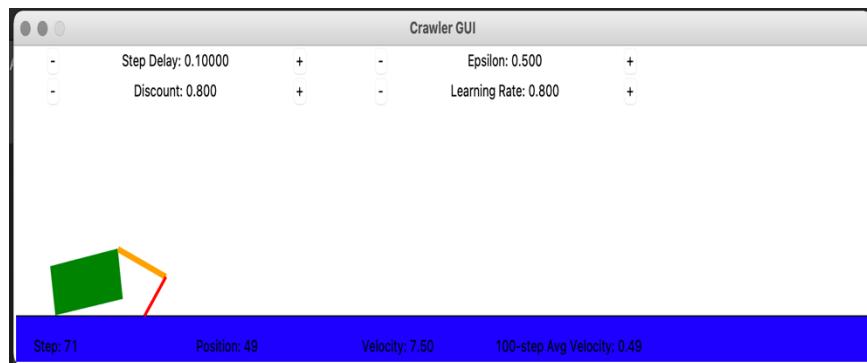
    if len(actions) == 0:
        return None
    else:
        if util.flipCoin(self.epsilon):
            action = random.choice(actions)
        else:
            action = self.getPolicy(state)

    return action
```

با استفاده از دستور `python gridworld.py -a q -k 100 -e 0.3` داریم:



با استفاده از دستور `python crawler.py` می توان یک ربات crawl یادگیری Q را اجرا کرد.

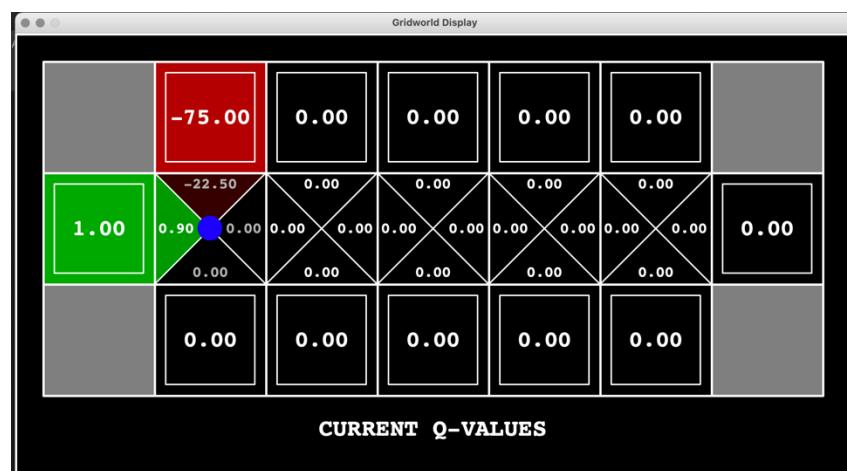


## بخش هشتم، بررسی دوباره عبور از پل

در این بخش پس از حدود ۵۰ تکرار QValue-iteration، سیاست بهینه باید برای مقادیر اپسیلون و نرخ یادگیری (با probability بالا) از پل عبور کند. در اصل این کار به آسانی امکان پذیر نمی باشد. متده question8 که نوشته ام رشته NOT POSSIBLE را بر می گرداند (به این معنی است که جوابی پیدا نشده است) زیرا برای اینکه یک سیاست بهینه با احتمال بیشتر از ۹۹ درصد، episode ۵۰ کوچک می باشد بنابراین ما به episode ۵۰ کوچک می باشیم. این بیشتری نیاز داریم.

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

در اصل با استفاده از دستور `python gridworld.py -a q -k 50 -n 0 -g BridgeGrid -e` (با اپسیلون صفر) خروجی زیر را داریم.



## بخش نهم، پک من و Q-Learning

همانطور که در دستور کار هم ذکر شد عامل ما بازی ها را در دو فاز آموزش (عامل ما در مرور امتیاز موقعیت ها و اکشن ها آموزش می بیند) فاز دیگر هم آزمون می باشد.

دستور زیر را اجرا کرده و خروجی را می بینیم:

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

```
Reinforcement Learning Status:  
    Completed 1700 out of 2000 training episodes  
    Average Rewards over all training: -116.64  
    Average Rewards for last 100 episodes: 276.44  
    Episode took 1.19 seconds  
Reinforcement Learning Status:  
    Completed 1800 out of 2000 training episodes  
    Average Rewards over all training: -100.31  
    Average Rewards for last 100 episodes: 177.15  
    Episode took 1.12 seconds  
Reinforcement Learning Status:  
    Completed 1900 out of 2000 training episodes  
    Average Rewards over all training: -77.27  
    Average Rewards for last 100 episodes: 337.55  
    Episode took 1.23 seconds  
Reinforcement Learning Status:  
    Completed 2000 out of 2000 training episodes  
    Average Rewards over all training: -60.05  
    Average Rewards for last 100 episodes: 267.11  
    Episode took 1.16 seconds  
Training Done (turning off epsilon and alpha)  
-----  
Pacman emerges victorious! Score: 503  
Pacman emerges victorious! Score: 499  
Pacman emerges victorious! Score: 499  
Pacman emerges victorious! Score: 503  
Pacman emerges victorious! Score: 495  
■ Average Score: 501.4  
Scores:      503.0, 503.0, 503.0, 503.0, 503.0, 503.0, 499.0, 499.0, 503.0, 495.0  
Win Rate:    10/10 (1.00)  
Record:     Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

دستور زیر را اجرا کرده و خروجی را می بینیم:

```
python autograder.py -q q9
```

## بخش دهم، Q-Learning تقریبی

در این بخش Learning تقریبی را پیاده سازی کرده ام. ابتدا آنچه که از ما خواسته شده است این است که باید بردار وزن را پیاده سازی کنیم، که ویژگی را به مقدار وزن ها نگاشت کند. در اصل برای هر feature که برای استیت و اکشن استخراج کردیم، وزن باید extract شود تا مقدار استیت را به ما دهد.

بنابراین در تابع getQValue معادله زیر را پیاده سازی کرده ام:

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

به این صورت که ابتدا مقدار QValue را صفر مقدار دهی کردم سپس با استفاده از extract ها feature .featExtractor.getFeatures(state,action) می کنیم. حال در حلقه تکراری که داریم معادله بالا را پیاده سازی کرده ام.

```
def getQValue(self, state, action):
    """
        Should return Q(state,action) = w * featureVector
        where * is the dotProduct operator
    """
    "*** YOUR CODE HERE ***"
    QValue = 0.0
    featureVector = self.featExtractor.getFeatures(state, action)
    for feature in featureVector:
        QValue += self.weights[feature] * featureVector[feature]
    return QValue
```

در متدهم باید بردارهای وزن را آپدیت کنیم. در اصل این متدهم از متدهای update و getQValue برای استیت فعلی از متدهای override و computevalueFromQvalue می‌شود و برای استیت بعد (nextState) با متدهای QValue و مقدار می‌گرفته شود. و معادلات زیر را باید پیاده سازی کنیم.

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

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

در کد ابتدامقدار QValue برای استیت فعلی و بعدی را به دست می‌آوریم. سپس difference را با توجه به معادله بالا می‌نویسیم. در لوب نوشته شده هم  $w_i$  را با توجه به معادله بالا حساب می‌کنیم.

```
def update(self, state, action, nextState, reward):
    """
        Should update your weights based on transition
    """
    *** YOUR CODE HERE ***
    QValueNextState = self.getValue(nextState)
    QValueCurrState = self.getQValue(state, action)
    difference = (reward + self.discount * QValueNextState) - QValueCurrState
    featureVector = self.featExtractor.getFeatures(state, action)

    for feature in featureVector:
        self.weights[feature] += self.alpha * difference * featureVector[feature]
```

با وارد کردن دستور `python pacman.py -p approximateqagent -x 2000 -n 2010 -l smallgrid` خروجی زیر حاصل می شود.

```
Reinforcement Learning Status:  
    Completed 1700 out of 2000 training episodes  
    Average Rewards over all training: -120.15  
    Average Rewards for last 100 episodes: 114.81  
    Episode took 1.52 seconds  
Reinforcement Learning Status:  
    Completed 1800 out of 2000 training episodes  
    Average Rewards over all training: -103.77  
    Average Rewards for last 100 episodes: 174.63  
    Episode took 1.68 seconds  
Reinforcement Learning Status:  
    Completed 1900 out of 2000 training episodes  
    Average Rewards over all training: -88.01  
    Average Rewards for last 100 episodes: 195.75  
    Episode took 1.90 seconds  
Reinforcement Learning Status:  
    Completed 2000 out of 2000 training episodes  
    Average Rewards over all training: -72.26  
    Average Rewards for last 100 episodes: 226.91  
    Episode took 1.59 seconds  
Training Done (turning off epsilon and alpha)  
-----  
Pacman emerges victorious! Score: 503  
Pacman emerges victorious! Score: 503  
Pacman emerges victorious! Score: 499  
Pacman emerges victorious! Score: 503  
Pacman emerges victorious! Score: 503  
Pacman emerges victorious! Score: 499  
Pacman emerges victorious! Score: 501  
Pacman emerges victorious! Score: 495  
Pacman emerges victorious! Score: 499  
Pacman emerges victorious! Score: 501  
Average Score: 500.6  
Scores:      503.0, 503.0, 499.0, 503.0, 503.0, 499.0, 501.0, 495.0, 499.0, 501.0  
Win Rate:    10/10 (1.00)  
Record:     Win, Win, Win, Win, Win, Win, Win, Win, Win, Win  
(venv) heliaa@MacBook-Pro reinforcement %
```

با وارد کردن دستور

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor  
-x 50 -n 60 -l mediumGrid
```

خروجی زیر حاصل می شود.

```
(venv) heliaa@MacBook-Pro reinforcement % python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor  
Beginning 0 episodes of Training  
Pacman emerges victorious! Score: 1323  
Average Score: 1323.0  
Scores: 1323.0  
Win Rate: 1/1 (1.00)  
Record: Win  
(venv) heliaa@MacBook-Pro reinforcement % -x 50 -n 60 -l mediumGrid
```

با وارد کردن دستور

```
python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor  
-x 50 -n 60 -l mediumClassic
```

خروجی زیر حاصل می شود.

```
Structure (venv) heliaa@MacBook-Pro reinforcement % python pacman.py -p ApproximateQAgent -a extractor=SimpleExtractor  
Beginning 0 episodes of Training  
Pacman emerges victorious! Score: 1522  
■ Average Score: 1522.0  
Scores: 1522.0  
Win Rate: 1/1 (1.00)  
Record: Win  
Favorites ★ (venv) heliaa@MacBook-Pro reinforcement % -x 50 -n 60 -l mediumClassic
```