强化学习 HW1

— 饶翔云 520030910366

Problem 1:

首先以数学形式重新阐述问题：

1. If we produce a policy π' from π, then $V^{\pi'} \geq V^{\pi}$（因为如果该公式成立，那么由于策略π'是根据V^{π}得来的，而$V^{\pi'} \geq V^{\pi}$，由$V^{\pi'}$所得到的策略肯定不比π'差）
2. If the optimal policy is π_{opt}, our policy will converges to it finally, and that means $V^{\pi_{opt}}$ is optimal with V^{π} will finally converges to it.(最优策略对应最大价值函数)

对于第一个问题：

当价值函数收敛后，对于任意状态s：

$V^{\pi'}(s) = \sum_{s'} P(s, \pi'(s), s')[R(s, \pi'(s), s') + \gamma V^{\pi'}(s')] = R(s, \pi'(s)) + \gamma \sum_{s'} P(s, \pi'(s), s') V^{\pi'}(s')$
$V^{\pi}(s) = \sum_{s'} P(s, \pi(s), s')[R(s, \pi(s), s') + \gamma V^{\pi}(s')] = R(s, \pi(s)) + \gamma \sum_{s'} P(s, \pi(s), s') V^{\pi}(s')$

从迭代公式易知：$V_0^{\pi'} = V^{\pi}$

又因为$\pi'(s) = argmax_a \sum_{s'} P(s, a, s')[R(s, a, s') + \gamma V^{\pi}(s')]$， 所以：

$V^{\pi}(s) \leq R(s, \pi'(s)) + \gamma \sum_{s'} P(s, \pi'(s), s') V_0^{\pi'}(s') = V_1^{\pi'}(s)$

故$V_1^{\pi'}(s) - V_0^{\pi'}(s) \geq 0$.假设已知$V_i^{\pi'}(s) - V_{i-1}^{\pi'}(s) \geq 0$, $\forall s$，那么：

$V_{i+1}^{\pi'}(s) - V_i^{\pi'}(s) = \gamma \sum_{s'} P(s, \pi', s')(V_i^{\pi'}(s') - V_{i-1}^{\pi'}(s')) \geq 0, \forall s$

所以$V_i^{\pi'}(s) \leq V_{i+1}^{\pi'}(s), \forall s \in S, \forall i \in \mathbb{N}$，

因此$V^{\pi'}(s) = V_{\infty}^{\pi'}(s) \geq V_0^{\pi'}(s) = V^{\pi}(s)$， 第一个问题证毕。

对于第二个问题：

$\forall i \in \mathbb{N}, \forall s \in S, |V_{\pi_{i+1}} - V_{\pi_i}| \leq \gamma |V_{\pi_i} - V_{\pi_{i-1}}|$待证

奖励函数有界，不妨令$R(s, a) \leq n$

那么$V^{\pi} = \mathbb{E}(\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_t') | \pi, s_0 = s) \leq n(1 + \gamma + \gamma^2 + \dots) = \frac{n}{1-\gamma}$

将$\{V^{\pi_i}\}$视为一个数列，那么根据单调递增有界收敛定理，$\{V^{\pi_i}\}$必定收敛。

那么$\{V^{\pi_i}\}$必有上确界，而这就是我们所求的最大价值函数，由这个最大价值函数得到的策略就是最优策略。

problem 2：

首先列举这个环境的性质：

- 二维的栅格世界
- 智能体有20%概率采取与理论不同的行为（ϵ-policy?）
- 智能体的目标是进入 `TERMINAL STATE` 并采取 `exit` 从而获得最终的奖励

(a) 实现基于价值迭代的智能体

共有三个部分的代码需要补全。

1. runValueIteration()

```
# implement the synchronous value iteration
        for _ in range(0, self.iterations):
            new_values = util.Counter()
            for state in self.mdp.getStates():
                # If the state is terminal, then the value is 0
                if self.mdp.isTerminal(state):
                    new_values[state] = 0
                else:
                # Otherwise, the value is the maximum Q of all possible
                #  actions(using old values)
                    actions = self.mdp.getPossibleActions(state)
                    new_values[state] = max(
                        [self.computeQValueFromValues(state, action)
                         for action in actions])
            # If the maximum change is less than epsilon, then stop
            change = [abs(self.values[state] - new_values[state])
                      for state in self.mdp.getStates()]
            if max(change) < self.epsilon:
                break
            # Update the values
            self.values = new_values
```

2. computeQValueFromValues()

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

        value = None

        """ YOUR CODE HERE """
        # The Q-value is the sum of all possible next states and
        # their probabilities
        value = 0
        for next_state, prob in self.mdp.getTransitionStatesAndProbs(state, action):
            value += prob * (self.mdp.getReward(state, action, next_state)
                             + self.discount * self.values[next_state])

        return value
```

3. computeActionFromValues()

```
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.
        """

        bestaction = None

        """ YOUR CODE HERE """
        # If the state is terminal, then there is no action
        if not self.mdp.isTerminal(state):
            # Find the action with the maximum Q
            actions = self.mdp.getPossibleActions(state)
            bestaction = max(actions, key=lambda action:
                             self.computeQValueFromValues(state, action))

        return bestaction

```

(b) 实现基于策略迭代的智能体

共有三个部分的代码需要补全。

1. runValueIteration()

```
def runPolicyIteration(self):
        """ YOUR CODE HERE """
        # calculate V according to the policy
        # initialize the policy
        for state in self.mdp.getStates():
            # If the state is terminal, then the policy is None
            if self.mdp.isTerminal(state):
                self.policy[state] = None
            else:
            # Otherwise, the policy is the one of the possible actions
                actions = self.mdp.getPossibleActions(state)
                idx = random.randint(0, len(actions)-1)
                self.policy[state] = actions[idx]
        current_policy = self.policy
        for _ in range(0, self.iterations):
            while True:
                new_values = util.Counter()
                for state in self.mdp.getStates():
                    # If the state is terminal, then the value is 0
                    if self.mdp.isTerminal(state):
                        new_values[state] = 0
                    else:
                    # Otherwise, the value is the maximum Q of all possible
                    # actions(using old values)
                        action = current_policy[state]
                        new_values[state] =
                            self.computeQValueFromValues(state, action)
                # If the maximum change is less than epsilon, then stop
                change = [abs(self.values[state] - new_values[state]) for state
                          in self.mdp.getStates()]
                # Update the values
                self.values = new_values
                if max(change) < self.epsilon:
                    break
            # calculate the new policy
            new_policy = dict()
            for state in self.mdp.getStates():
                # If the state is terminal, then the policy is None
                if self.mdp.isTerminal(state):
                    new_policy[state] = None
                else:
                # Otherwise, the policy is the action with the maximum Q
                    actions = self.mdp.getPossibleActions(state)
                    new_policy[state] = max(actions,
                    key=lambda action: self.computeQValueFromValues(state, action))
            # If the new policy is the same as the old policy, then stop
            if new_policy == current_policy:
                break
            else:
                current_policy = new_policy
        # Update the policy
        self.policy = current_policy
```

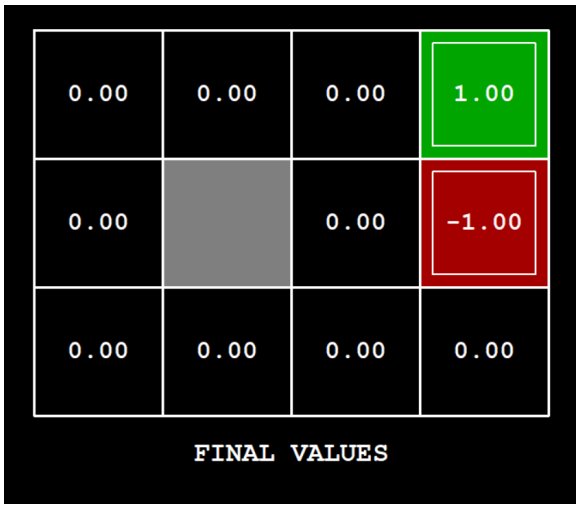
2. computeQValueFromValues()
同基于价值迭代的智能体。

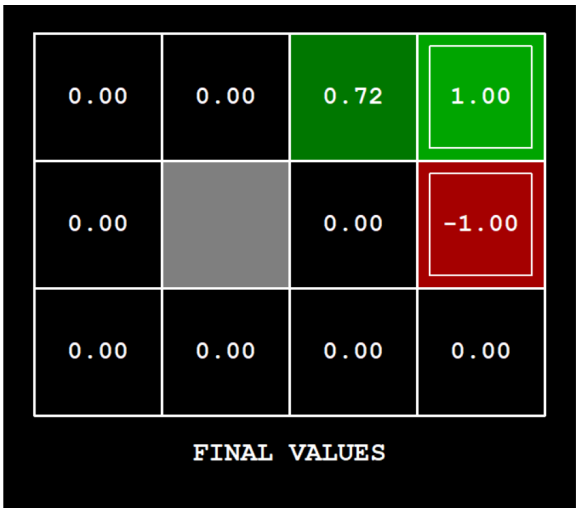
3. computeActionFromValues()
同基于价值迭代的智能体。

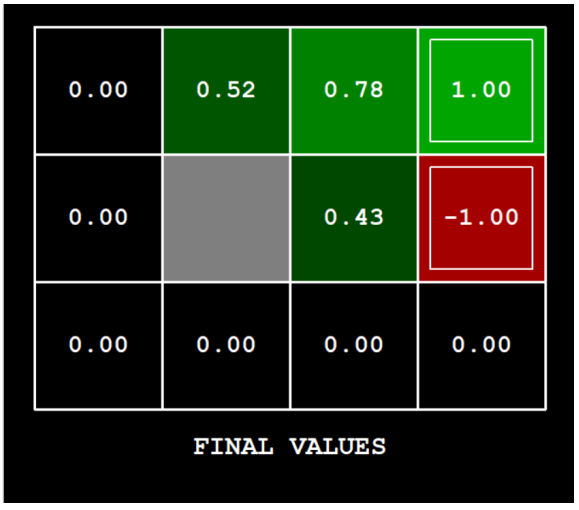
(c) 画图，探究收敛速度并说出原因

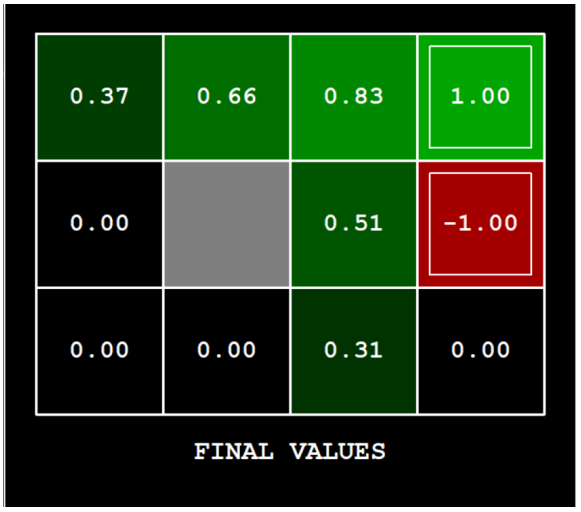
我将ϵ改为0.1，理由是更能看出收敛速度的差别。

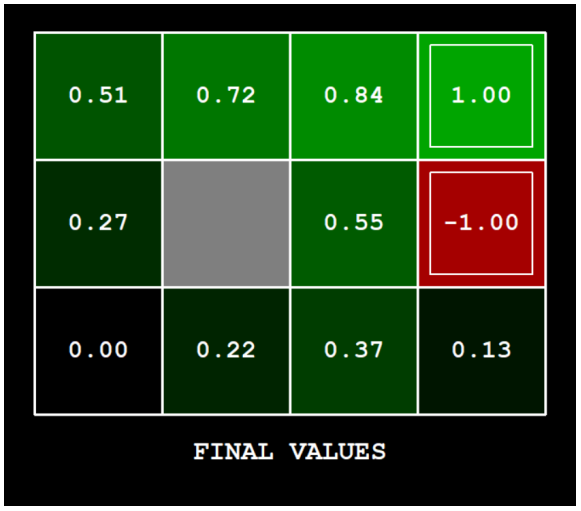
价值迭代收敛后结果如下：

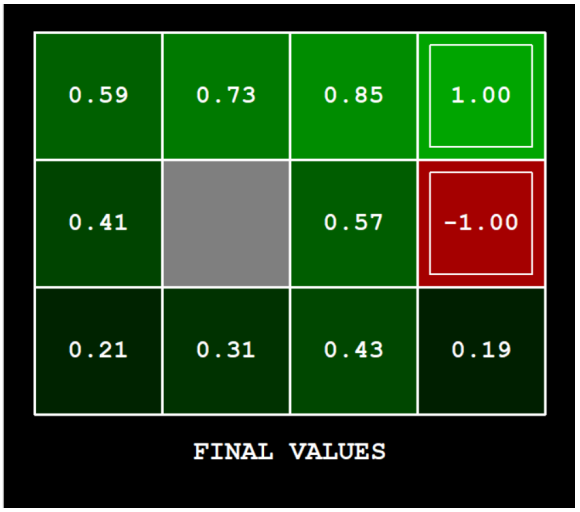


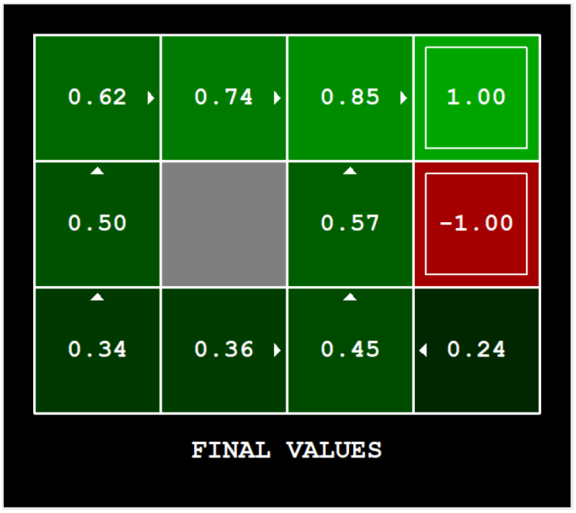


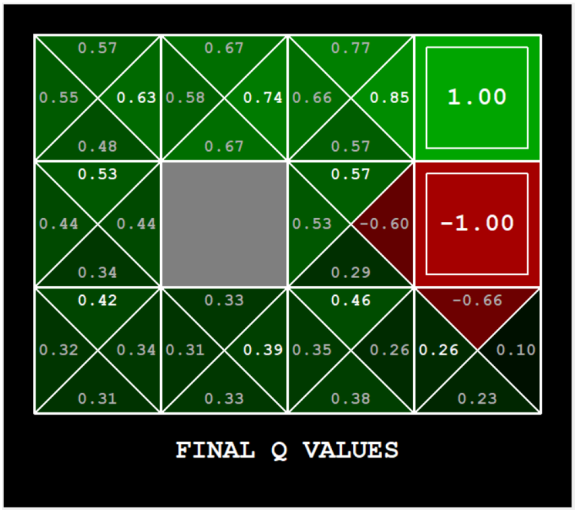






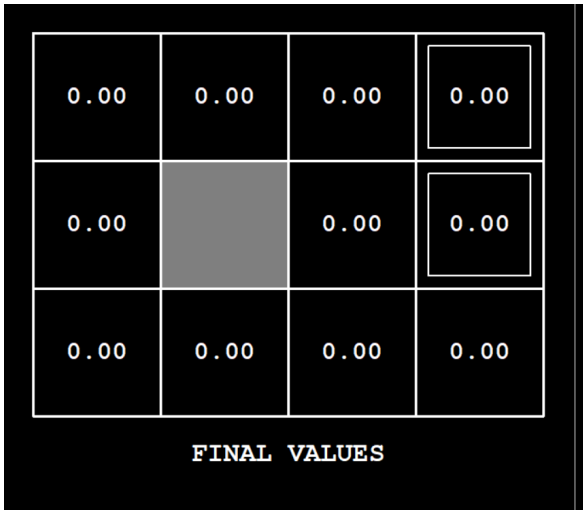


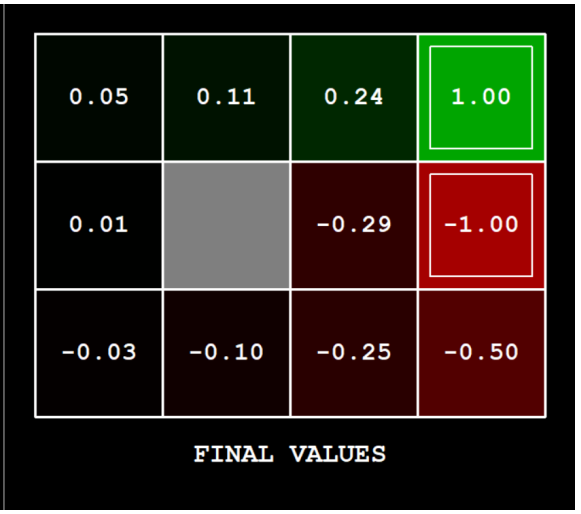


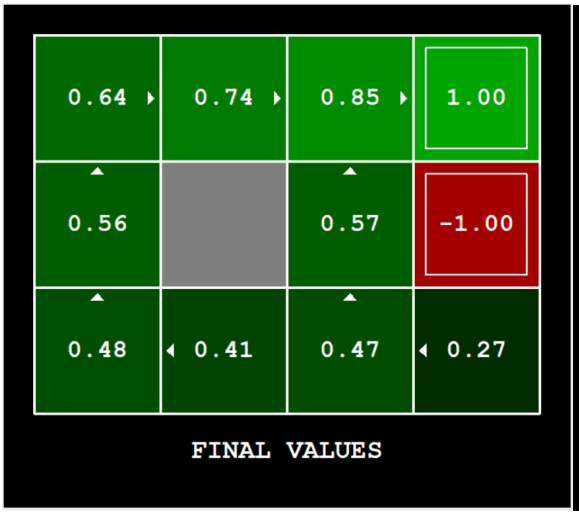


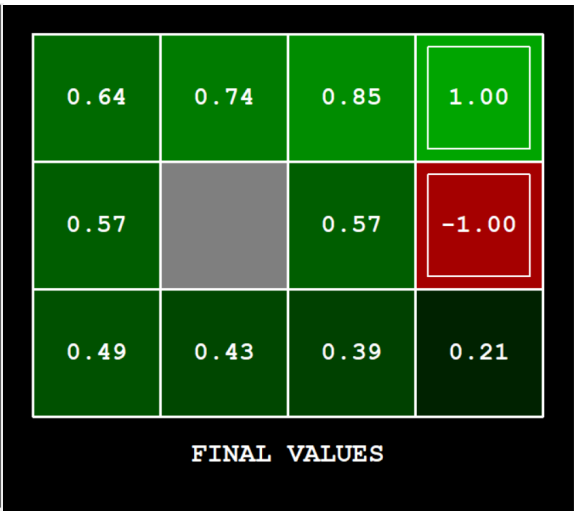
经过探究，V和Q在第7轮完全收敛，花了0.002秒（在0.001~0.003秒浮动）

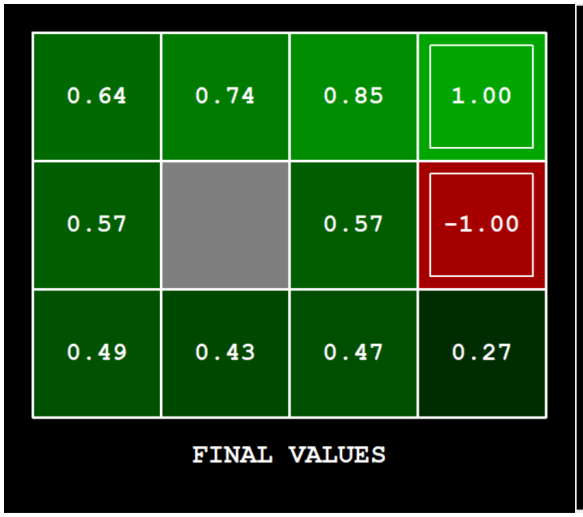
策略迭代收敛后结果如下：

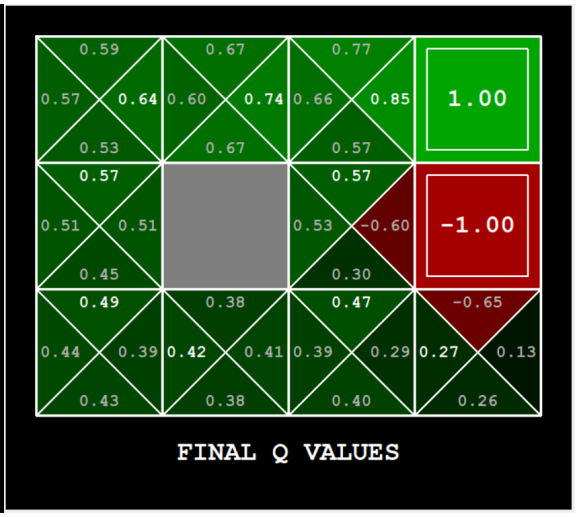








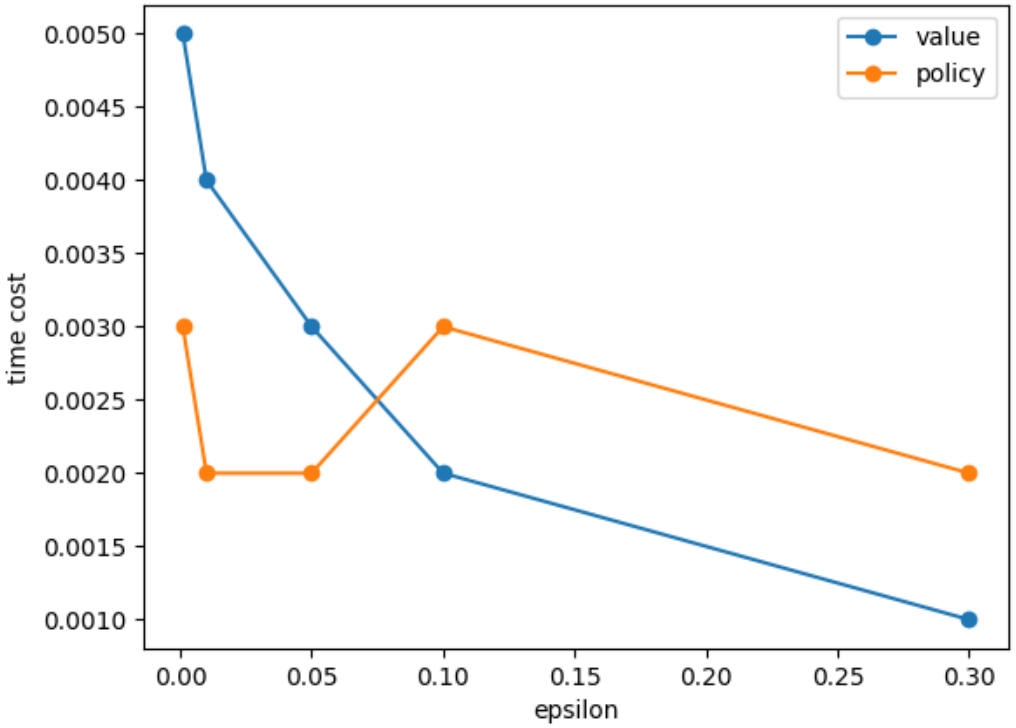




经过探究，V和Q在第4轮就已经完全收敛，花了0.003秒（在0.002~0.004秒浮动）

将ϵ改为0.01后，发现策略迭代需要0.002秒，而价值迭代需要0.004秒。

设置$\epsilon \in [0.001, 0.01, 0.05, 0.1, 0.3]$,画出统计图如下：



结论：在这个gridworld中，当ϵ比较小的时候，策略迭代收敛速度更快;当ϵ比较大的时候，价值迭代更快。

原因：这个结论的前提是在这个gridworld中，因为这个gridworld相对较小，很容易找到最优策略，策略迭代和价值迭代时间开销才可比较。因为策略迭代在每一轮迭代都会进行一次价值迭代（基于当前策略）。当ϵ比较小的时候，价值函数要收敛花费时间长，价值迭代轮数多。但是策略迭代的每一轮都将价值迭代中贪心求最优动作的步骤改为基于当前策略取动作，这一步节省的开销由于迭代轮数变多从而减小了总时间开销。当ϵ比较大的时候，价值迭代轮数少，策略迭代每轮的价值迭代所节约的开销就减少了，而要策略要迭代的轮数还在那，从而时间开销比正常的价值迭代更大。