**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**

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.