UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Probleme de cautare si agenti adversariali

Inteligenta Artificiala

Autori: Istrate Cristian Andrei si Ivan Alexandru Sofron
Grupa: 30237

Facultatea de Automatica
si Calculatoare

16 Ianuarie 2024

Cuprins

1 Reinforcement Learning

1.1 Question 1 - Value iteration

Ca prima cerinta am avut de realizat un "Value iteration agent", care este un planificator offline asa ca ceea ce ne intereseaza pe noi este ca in functie de fiecare iteratie pe care o face spre finalul labirintului sa asociem mutarilor cate o valoare in functie de cat de aproape suntem de final. La fiecare final de joc, valorile se actualizeaza in functie de cat de buna a fost mutarea spre finalul jocului.

Cheia spre implementarea acestui algoritm a fost ecuatia Bellman (figura 1). Astfel am avut de implementat 3 functii care au dus la construirea si aplicarea algoritmului Bellman asupra problemei noastre.

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

Figura 1: Ecuatia Bellman

```
    def runValueIteration(self):
        # Write value iteration code here
        "* YOUR CODE HERE *"
        for _ in range(self.iterations):
            new_values = util.Counter()
            for state in self.mdp.getStates():
                if not self.mdp.isTerminal(state):
                    new_values[state] = max(self.computeQValueFromValues(state, action)
                    for action in self.mdp.getPossibleActions(state))
            self.values = new_values

    def computeQValueFromValues(self, state, action):
        """
          Compute the Q-value of action in state from the
          value function stored in self.values.
        """
        "* YOUR CODE HERE *"
        q_value = 0
        for next_state, prob in self.mdp.getTransitionStatesAndProbs(state, action):
            reward = self.mdp.getReward(state, action, next_state)
            q_value += prob * (reward + self.discount * self.values[next_state])
        return q_value
        # util.raiseNotDefined()

    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 *"
          possible_actions = self.mdp.getPossibleActions(state)

          if not possible_actions:
              return None

          best_q_value = float('-inf')
          best_action = None

          for action in possible_actions:
              q_value = self.computeQValueFromValues(state, action)
              if q_value > best_q_value:
                  best_action = action
                  best_q_value = q_value

          return best_action
          # util.raiseNotDefined()
```

Pentru a reusi implementarea functiei Bellman a trebuit sa intelegem ecuatia Bellman si sa o impartim in bucati pentru a putea implementa algoritmul de care aveam nevoie.

1.2 Question 3 - Policies

Pentru a doua cerinta am avut de stabilit valori pentru discount, noise si living reward astfel incat sa parcurgem labirintul in cele 5 cazuri. Cele 3 valori semnifica:

- answerDiscount: Reprezinta cat de mult ne concentram pe viitoarele recompense, mai mic insemnand ca mergem pe cele mai apropiate recompense, iar mai mare pe cele departate.
- answerNoise: Reprezinta probabilitatea de efectua o miscare aleatoare, diferita de miscarea aleasa de noi.
- answerLivingReward: Reprezinta cat de mult ne dorim sa ramanem intr-o stare, astfel incrementand aceasta variabila, petrecem mai mult timp fara sa incheiem jocul.

```
python gridworld.py -g DiscountGrid -a value --discount [YOUR_DISCOUNT] --noise [YOUR_NOISE] --livingReward [YOUR_LIVING_REWARD]
```

Figura 2: GUI command

Dupa multiple teste cu comanda din figura 2 am ajuns la urmatoarele valori:

```
def question3a():
    answerDiscount = 0.1
    answerNoise = -0.1
    answerLivingReward = 0.0
    return answerDiscount, answerNoise, answerLivingReward
    # If not possible, return 'NOT POSSIBLE'
```

```
7
8  def question3b():
9      answerDiscount = 0.1
10     answerNoise = 0.1
11     answerLivingReward = 0.0
12     return answerDiscount, answerNoise, answerLivingReward
13     # If not possible, return 'NOT POSSIBLE'
14
15 def question3c():
16     answerDiscount = 0.5
17     answerNoise = 0.0
18     answerLivingReward = 0.0
19     return answerDiscount, answerNoise, answerLivingReward
20     # If not possible, return 'NOT POSSIBLE'
21
22 def question3d():
23     answerDiscount = 0.5
24     answerNoise = 0.1
25     answerLivingReward = 0.0
26     return answerDiscount, answerNoise, answerLivingReward
27     # If not possible, return 'NOT POSSIBLE'
28
29 def question3e():
30     answerDiscount = 0.0
31     answerNoise = 0.0
32     answerLivingReward = 1.0
33     return answerDiscount, answerNoise, answerLivingReward
34     # If not possible, return 'NOT POSSIBLE'
```

1.3 Question 5 - Q-Learning

Pentru ultima cerinta am avut de completat mai multe metode ale clasei "QLearningAgent". Acest agent nu mai foloseste modelul MDP pe care il folosea "Value iteration" ci se bazeaza pe modelul de invatare prin incercare, astfel se foloseste de mediul in care se afla pentru a-si calcula miscarile. Pentru a reusi implementarea acestor metode ne-am folosit de clasa "ValueIteration" pe care am implementat-o la Q1 pentru a avea un punct de plecare.

Metodele care difera de Q1 sunt: computeValueFromQValues si update. Prima metoda a fost implementata cu ajutorul comentariilor oferite ca suport, iar cea de a doua reprezenta formula de baza pentru a implementa Q learning agent si a fost necesar sa cautam mai multe informatii despre acest agent, intr-un final gasind formula care a dus la rezolvarea functiei (Fig. 3).

$$Q^{new}(S_t, A_t) \leftarrow (1 - \underbrace{\alpha}_{\text{learning rate}}) \cdot \underbrace{Q(S_t, A_t)}_{\text{current value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \underbrace{\left(\underbrace{R_{t+1}}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(S_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}_{\text{new value (temporal difference target)}}$$

Figura 3: Q Learning equation

```
def _init_(self, **args):
        "You can initialize Q-values here..."
        ReinforcementAgent._init_(self, **args)
        "* YOUR CODE HERE *"
        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
        """
        "* YOUR CODE HERE *"
        return self.qValue[(state, action)]
        # util.raiseNotDefined()


    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 *"
        legalActions = self.getLegalActions(state)
        if not legalActions:
            return 0.0
        return max(self.getQValue(state, action) for action in legalActions)
        # util.raiseNotDefined()

    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 *"
        legalActions = self.getLegalActions(state)
        if not legalActions:
            return None

        maxQValue = float('-inf')
        bestAction = None

        for action in legalActions:
            qValue = self.getQValue(state, action)
            if qValue > maxQValue:
```

```
49                  maxQValue = qValue
50                  bestAction = action
51
52          return bestAction
53          # util.raiseNotDefined()
54
55      def update(self, state, action, nextState, reward):
56          """
57            The parent class calls this to observe a
58            state = action => nextState and reward transition.
59            You should do your Q-Value update here
60
61            NOTE: You should never call this function,
62            it will be called on your behalf
63          """
64          "* YOUR CODE HERE *"
65          currentQValue = self.getQValue(state, action)
66          nextQValue = self.computeValueFromQValues(nextState)
67
68          updatedQValue = (1 - self.alpha) * currentQValue + self.alpha * (reward +
69          self.discount * nextQValue)
70          self.qValue[(state, action)] = updatedQValue
71          # util.raiseNotDefined()
```