UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Reinforcement learning

Inteligenta Artificiala

Darius-Iulian Stan
Grupa: 30231

Facultatea de Automatica
si Calculatoare

18 Ianuarie 2023

Cuprins

1 Introducere

În cadrul acestui proiect, în primă fază, implementăm Value iteration și analizăm parametrii modelului MDP (Markov Decision Process), el fiind caracterizat astfel:

- Un set de stări S
- Un set de acțiuni A (pentru fiecare stare din S)
- O funcție de traziție T(s,a,s'), care reprezintă probabilitatea ca, din starea s, dorind să facem acțiunea a, ajungem în starea s'
- o fucție de recompense R(s,a,s'): recompense pentru că agentul încă este în viață (pot fi negative) și recompense de terminare a jocului (ori foarte mari, ori foarte mici, depinzând de caracterul stării de finish)
- O stare inițiala și, eventual, una de exit

Ne aflăm într-un mediu non-deterministic, asta însemnând că daca alegem să facem acțiunea a, nu este garantat că outcomu-ul acesteia este cel așteptat. (Spre exemplu, dacă vrem să mergem din poziția curentă în Nord, s-ar putea, de fapt, să ajungem în poziția din Est). Astfel, definim zgomotul ("noise") ca fiind probabilitatea ca agentul să facă o altă acțiune decât cea cerută.

Formula de bază a acestui proiect este cea a lui Bellman, care definește valoarea optimală a unei stări ca fiind maximul dintre valorile Q-stărilor ce pot fi atinse făcând orice acțiune posibilă din starea dată. Q-valoarea, la rândul ei, este definită, pentru o stare s și o acțiune a, calculând suma produsurilor dintre probabilitatea de a ajunge într-o anumită stare s' în urma efectuării acțiunii a în starea s, și suma recompensei acestei tranziții (s-a-s') cu valoarea stării respective* un discount anume.

2 Q1: Value Iteration

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

Pe baza formulei de recurență de mai sus, provenind din formula lui Bellman caracterizand funcția pentru valori optimale, am implementat metodele: computeActionFromValues, computeQValueFromValues, runValueIteration din clasa ValueIterationAgent.

cod:

```
    def runValueIteration(self):

        states = self.mdp.getStates()
        for iteration in range(self.iterations):
            current_values = util.Counter()
            for s in states:
                if not self.mdp.isTerminal(s):
                    current_values[s] = self.computeQValueFromValues(s,self.getAction(s))
            self.values=current_values

def computeQValueFromValues(self, state, action):

        transitions = self.mdp.getTransitionStatesAndProbs(state,action)
        if transitions is None:
            return 0
        Qvalue = 0
```

```
17          for(s, T) in transitions:
18              R = self.mdp.getReward(state,action,s)
19              Qvalue +=T*(R + self.discount*self.values[s])
20          return Qvalue
21
22      def computeActionFromValues(self, state):
23
24          if self.mdp.isTerminal(state):
25              return None
26          best = {"action": None, "value": None}
27          actions = self.mdp.getPossibleActions(state)
28          for action in actions:
29              qVal = self.computeQValueFromValues(state,action)
30              if best["action"] is None or best["value"] < qVal:
31                  best = {"action": action, "value": qVal}
32          return best["action"]
```

în self.values avem valorile pentru tabla noastră, ele inițial fiind zero (nu s-a efectuat nicio iterație, k = 0). Astfel, la fiecare iterație, pentru fiecare stare, dacă nu este terminală, actualizăm valoarea sa coform formulei, folosind datele de la iterația anterioară.

computeActionFromValues returnează cea mai bună acțiune pe care o pot face din starea dată (state), iar computeQValueFromValues crează suma din formula dată, pentru o aumită stare curentă si acțiune dată. Astfel, în runValueIteration, apelând computeQValueFromValues pentru starea curentă si acțiunea returnată de computeActionFromVaues, nu mai trebuie să calculăm maximul dintre aceste sume, întrucât facem suma stric pentru cea mai bună acțiune.

3 Q2: Bridge Crossing Analysis

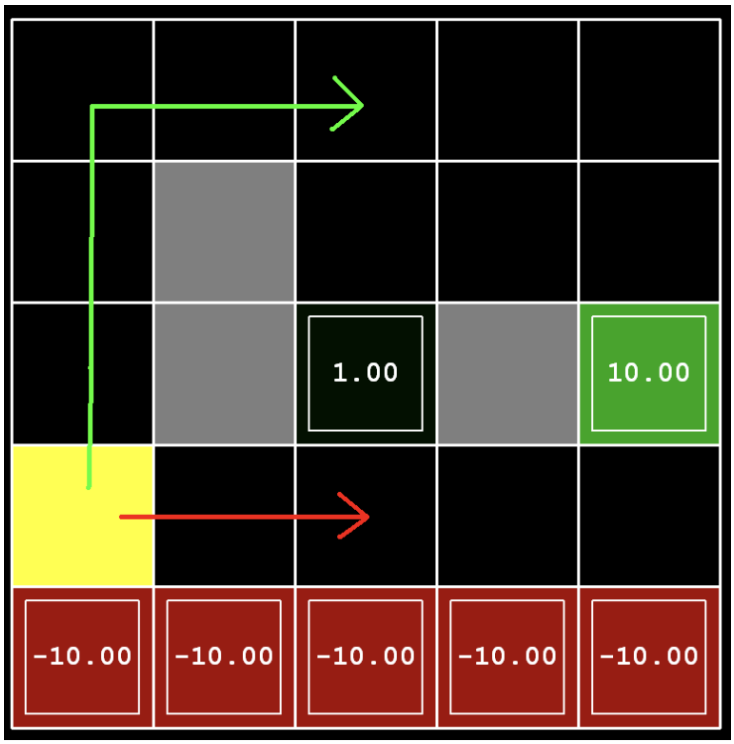
În cadrul acestui task, se dă următorul grid, reprezentând un "pod", trebuind să modificăm doar unul dintre parametrii dați (noise - probabilitatea să fac altă acțiune decât cea aleasă si discount - gamma), ei fiind noise = 0.2 și discount = 0.9.

Confrom, formulei, pentru ca, din orice stare componentă podului, pentru a a-l traversa cu succes, dacă noise-ul ar fi 0.0, astfel că mediul ar fi unul deterministic, în care fiecare acțiune

duce la starea aleasă, evident că podul ar fi traversat cu succes (presupunând că discount-ul este totuși pozitiv, întrucât unul negativ ar inversa polaritatea valorilor optimale în formula Q-valorilor, astfel că cele mai desirabile ar fi cele cu -100, podul ne mai fiind traversat). Totuși, din mai multe încercări, am observat că pentru discount-ul de 0.9, podul este traversat cu valoarea lui noise fiind dată din intervalul [0.0, 0.016].

4 Q3: Policies

Asemănător cu task-ul anterior, dat fiind grid-ul alăturat, trebuie să dăm valori parametrilor (noise, discount, livingReward) astfel încât să avem urmoarele cazuri:



- Preferă finalState-ul 1.00, riscând să cazi în prăpastie: evident, pentru ca din patratul galben, să aleg ruta spre Est, în detrimentul celei din Nord, voi alege un noise extrem de mic, iar pentru a merge la cei mai apropiat finalState, setând livingReward-ul la o valoare negativă (-1, în cazul nostru), agentul nu fa vi tentat să meargă până la finalState-ul 10.00, întrucât livingReward-ul este unul negativ, iar discount-ul, fiind și el mic, nu are cum să echivaleze (produsul dintre el si 10.00) valorile cumulate de livingRewards.
- Preferă finalState-ul 1.00, dar evitând prăpastia: pentru a alege ruta spre Nord, este suficient ca noise-ul să fie 0.3 (dacă este mai mare, atunci, cel mai probabil, ajungem într-o altă stare decât cea dorită, deci agentul va alege chiar acțiunea opusă, pentru a crește șansele să merg în direcția mea dorită inișial), livingReward-ul este pus pe -3, însă o valoare mai mare (în modul) poate duce la decizia agentului de a se risca sa parcurgă calea către 1 de lângă prăpastie. Totodată, discount-ul de 0.4 ajută pentru echilibrarea livingRewardului, întrucât influențează agentul să meargă totuși spre un finalState pozitiv, garantând că scorul final va fi mai mare decât -10.00 (dacă s-ar arunca în prăpastie direct)
- Preferă FinalState-ul 10.00, riscând prăpastia: Asemănător primul scenariu, noise-ul trebuie să fie unul mic, iar discount-ul unul mare (0.9) astfel încât să fie preferată ruta care duce cel mai rapid spre finalState. Mai mult, livingResult-ul trebuie să fie îndeajuns de

mare încât să evite finalState-ul 1.00, însă nu prea mare pentru a eclipsa discount-ul și a alege ruta spre Nord.

- Preferă FinalState-ul 10, dar fără a risca prăpastia: LivingReward-ul unul mare, astfel încât agentul să dorească să viziteze cât mai multe stări, Noise-ul unul mediu (0.4), pentru a nu se dori parcurgerea căii de lângă prăpastie, iar discount-ul trebuie să fie destul de mare astfel încât să ajung, totuși, în finalState 20.0.
- : jocul nu se termină niciodată: evident, livingRewardul trebuie să eclipseze produsul dintre discounted si valorile finalState-urilor pozitive. Deci, livingState-ul-o valoare mare -, discount-ul unul mic, aproape de 0 (sau chiar 0). Noise-ul trebuie să fie, la fel, 0.4, pentru a nu se risca alegerea căii de lângă prăpastie, pentru că ar risca terminarea jocului (căderea în infern).

5 Q5: Q-Learning

Dat fiind că, în contextul subpunnctelor anterioare, agentul nostul nu învață pe baza mediului în care se află în mod treptat, ci își crează harta acțiunilor corespunzătoare stărilor bazată pe valorile V(s), lucru ce în lumea reală nu este posibil, pentru a crea totuși o strategie în care agentul să experimenteze fiecare stare și pontențialele sale outcome-uri, implementam Q-learning-ul. După cum îi spune și numele, el este bazat pe q-state-uri, nu pe valorile V ale stărilor propriu zise. Astfel, nu știm cu siguranță dacă o acțiune este întradevăr desirabilă sau nu, însă, prin efectuarea unui număr mare de iterații, cresc șansele de a primi toate variantele posibile pentru un Q-state, predominând, evident, cele cu probabilitatea mai mare. Astfel, pornind de la formula lui BellMan pentru Q-value, obținem:

$$Q(s,a) \leftarrow (1-\alpha)\cdot Q(s,a) + \alpha \cdot \left(r + \gamma \cdot \max_{a'} Q(s',a')\right) \tag{1}$$

Pentru o vizibilitate mai sporită, vom nota suma din paranteză cu sample.

astffel, din formula de mai sus putem observa ca, cu ajutorul lui alpha, care reprezintă rata de învățare, definim q-state-ul ca fiind media dintre valoarea pe care o avea deja din outcome-urile precente și outcome-ul nou găsit (sample). Practic, Q-state ul este de fapt media tuturor outcome-urilor găsite până în momentul de față cu noul outcome.

Astfel, în clasa QLearningAgent, am implementat metodele: computeValueFromQValues (returnează cea mai bună valoare a Q-state-urilor din starea s, pentru fiecare acțiune posibilă), computeActionFromQValues(returnează cea mai bună acțiune din state-ul dat), iar update actualizează Q-valoarea (state,action) conform formulei recursive de mai sus.

```
1      def getQValue(self, state, action):
2          return self.Qvalue[(state, action)]
3
4
5      def computeValueFromQValues(self, state):
6
7          actions = self.getLegalActions(state)
8          if not actions:
9              return 0
10         valueQAS = max([self.getQValue(state,action) for action in actions])
11         return valueQAS
12
```

```
    def computeActionFromQValues(self, state):

        actions = self.getLegalActions(state)
        if not actions:
            return 0
        action = max(actions, key=lambda action: self.getQValue(state, action),
                 default=None)
        return action
    def update(self, state, action, nextState, reward):

    sample = reward + self.discount*self.getQValue(nextState,
             self.computeActionFromQValues(nextState))
    self.Qvalue[(state, action)] = (1 - self.alpha)*self.getQValue(state, action) +
                                    self.alpha*(sample)
```

6 Q6: Epsilon Greedy

Pentru ca lumea în care antrenăm agentul să fie cât mai imprevizibilă, construim metoda getAction() în următorul fel: pentru starea in care ne aflam, alegem o actiune in od aleatoriu in epsilor la suta din cazuri, in celelalte cazuri, selectam cea mai buna actiune disponibila din starea respectivă, alegere făcută pe cunoștințele dobândite până în prezent.

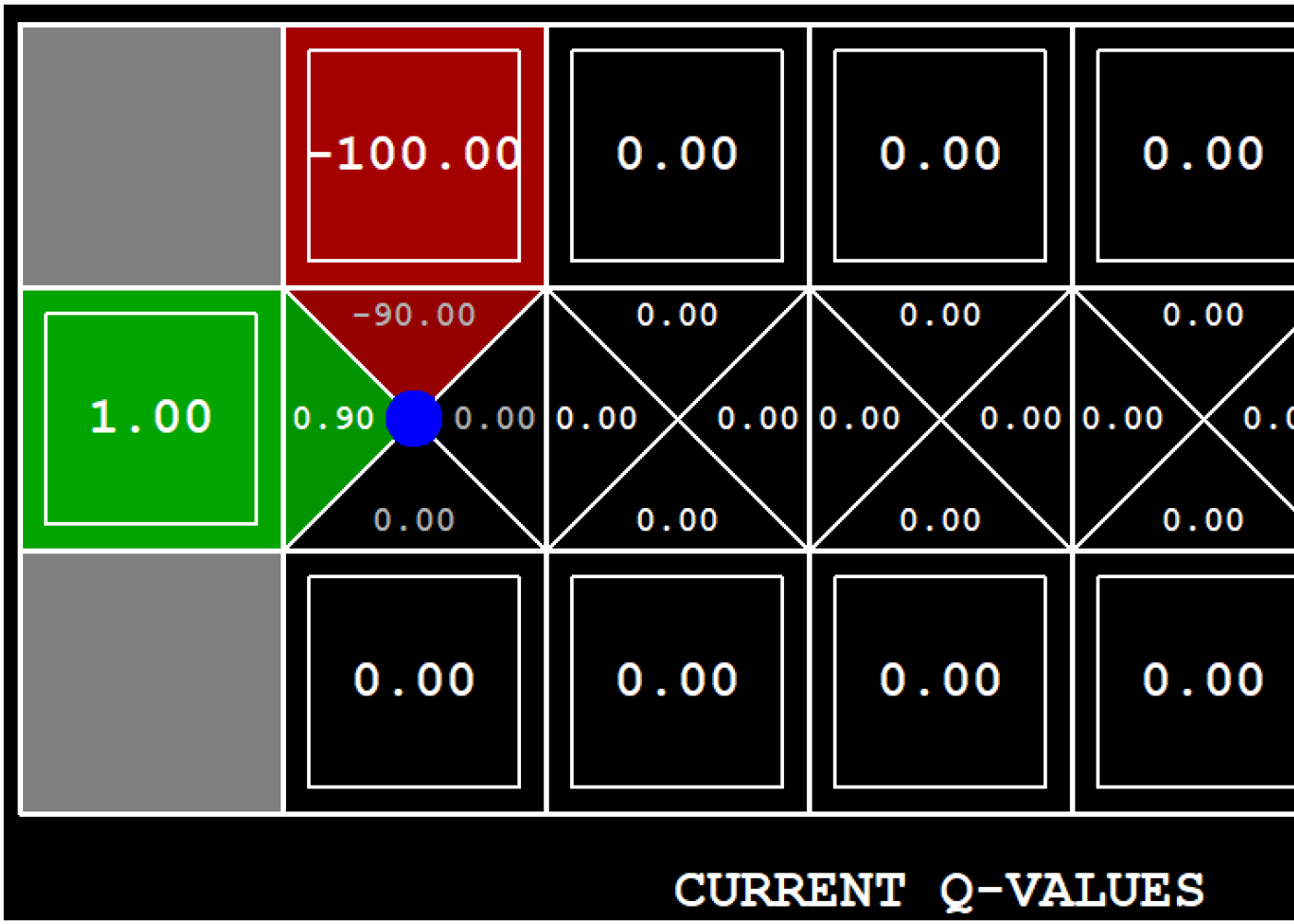
```
    def getAction(self, state):
        # Pick Action
        legalActions = self.getLegalActions(state)
        action = None

        if not legalActions:
            return None
        if util.flipCoin(self.epsilon):
            action = random.choice(legalActions)
        else:
            action = self.getPolicy(state)

        return action
```

7 Q7: Bridge Crossing Revisited

Revenind la problema podului din Q2, vrem să verificăm dacă, pentru Q-learning, există valori pentru epsilon si alpha (learning rate) astfel încât un anumit grid (cel din imagine), să conveargă dupa 50 de k iterații. Însă, răspunsul este negativ! Dat fiind ca epsilor de câte ori fac o acțiune neprevăzută, afectează direct proporțional spațiul stărilor care sunt vizitate (de exemplu, pentru epsilor 0, la un moment dat va exista bucla continuă de Est - West dintre casutele). Dacă epsilor este 1, atunci, întradevăr, spațiul stărilor exporate se extinde, însă nu îndeajuns astfel încât să se acopere, cu valorile corespunzătoare, fiecare celulă.



8 Q8: Q-Learning and Pacman

Cu ajutorul Q-learn-ului, Pacman reușește, după 2000 de teste pentru învățare a mediului, să aplice cu succes politica lui bazată de experienă, evitând fantomele și luând recompensa.