Umjetna inteligencija

Podržano učenje

Marko Čupić

IZDAVAČ

JAVNO DOSTUPNO NA WEB STRANICI JAVA.ZEMRIS.FER.HR/NASTAVA/UI

Ovo je popratni materijal za kolegij Umjetna inteligencija na Fakultetu elektrotehnike i računarstva Sveučilišta u Zagrebu. Tekst je usklađen s prezentacijama koje se koriste na tom kolegiju i okvirno ih prati. Umjesto kroz pseudokodove, primjeri su ilustrirani konkretnim programskim kodom u programskom jeziku Java. Uz tekst je pripremljen i dodatni Eclipse-projekt (objavljen kao ZIP-arhiva) s cjelovitom programskom implementacijom svih algoritama koji se spominju u tekstu zajedno s dva primjera. Čitatelj se upućuje da skine taj projekt i prati te pokreće primjere o koji se obrađuju u tekstu.

Prvo izdanje, travanj 2020.

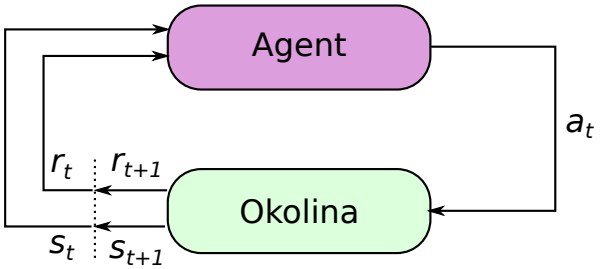
Sadržaj

1. Uvod

Razvoj računalnih agenata koji su u stanju donositi optimalne odluke u danom kontekstu težak je problem, a njegove začetke možemo pratiti već od 60-tih godina dvadesetoga stoljeća. Ovisno o načinu na koji problem možemo formalno opisati razvijene su i prikladne tehnike za njegovo rješavanje.

Primjerice, ako okolinu u kojem agent djeluje možemo opisati propozicijskom logikom, tada zaključivanje možemo provoditi nekim od pravila zaključivanja (primjerice, rezolucijskim pravilom) i na temelju zaključaka možemo donositi odluke. Ako okolina nije toliko jednostavna da bismo je mogli opisati propozicijskom logikom, ali jest dovoljno jednostavna da bismo je mogli opisati predikatnom logikom, i tada nam na raspolaganju stoje odgovarajući mehanizmi zaključivanja, ali i nezgodan problem: predikatna logika je poluodlučiva –dokazivanje istinitih tvrdnji možemo obaviti u konačnom broju koraka, no dokazivanje da je lažna tvrdnja doista lažna može nikada ne završiti.

U okviru ove teme razmatrat ćemo vremenski diskretne okoline u kojima u trenutku t agent percipira stanje okoline $s_t \in \mathscr{S}$, na temelju te percepcije bira jednu od akcija $a_t \in \mathscr{A}_t$ koja mu u tom stanju stoji na raspolaganju te odabranu akciju izvodi. Ako s $\mathscr{A}$ označimo skup svih mogućih akcija koje agent poznaje, tada oznaka $\mathscr{A}_t$ predstavlja podskup tog skupa, odnosno skup svih akcija koje agentu stoje na raspolaganju u trenutku t, odnosno kad je okolina u stanju s_t. Izvođenjem odabrane akcije agent djeluje na okolinu u kojoj se nalazi i kao posljedicu toga u trenutku $t+1$



Slika 1.1: Model okoline koji razmatramo

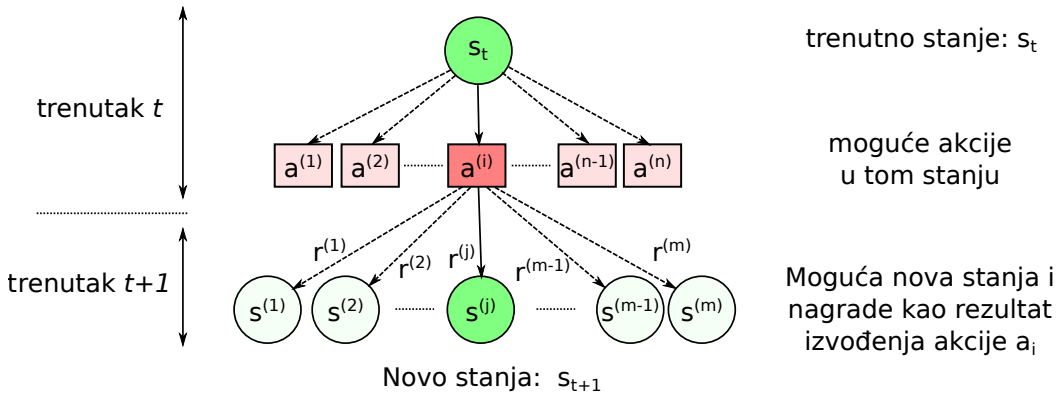
percipira neko novo stanje okoline s_{t+1} i od okoline dobiva određenu nagradu $r_{t+1} \in \mathbb{R}$. Neka od stanja iz $\mathscr{S}$ mogu biti terminalna stanja. Ako takva postoje, iz njih više nije moguće obavljati nikakve akcije i ona modeliraju situacije u kojima je agent obavio svoj zadatak ili više ne može izvoditi nikakve akcije (primjerice, jer je uništen). Opisan model okoline i interakcije s agentom prikazan je na slici 1.1.

Sučelje `DiscreteEnvironment` predstavlja programski model ovakve diskretne okoline i dano je u nastavku. Parametar `S` predstavlja tip stanja, a parametar `A` tip akcije. Metodom `getCurrentState` agent može doznati u kojem se stanju nalazi okolina. Metoda `setCurrentState` omogućava postavljanje trenutnog stanja okoline (primjerice, resetiranje u početno stanje). Metodom `getAvailableActions` agent može doznati koje mu akcije stoje na raspolaganju u trenutnom stanju okoline. metodom `applyAction` agent djeluje na okolinu obavljajući predanu akciju; kao rezultat tog djelovanja okolina mu vraća određenu nagradu i interno mijenja trenutno stanje. Metodom `isFinished` agent može doznati je li riješio zadatak (drugim riječima, je li okolina prešla u terminalno stanje).

Pseudokod 1.1 — Diskretna okolina.

```
package rl.model;

import java.util.Set;

public interface DiscreteEnvironment<S, A> {
  S getCurrentState();
  void setCurrentState(S currentState);
  Set<A> getAvailableActions();
  double applyAction(A action);
  boolean isFinished();
}
```

Kada agent odluči nad okolinom napraviti određenu akciju, dozvolit ćemo vrlo općenitu situaciju u kojoj rezultat izvođenja te akcije može biti nedeterministički. Konkretno, zamislite rešetkasti svijet sastavljen od ćelija po kojima se kreće robot s dva kotača. Neka robot može pokrenuti jednu od četiri akcije: naprijed, natrag, lijevo, desno. Pretpostavimo da je pokrenuo akciju "naprijed", zbog koje je istovremeno aktivirao svoj lijevi i desni kotač. Kako se kotači okreću istom brzinom, očekivali bismo da će nakon nekog vremena robot prijeći u susjednu ćeliju koja je



Slika 1.2: Djelovanje akcije na okolinu

bila ispred njega. Međutim, ako je površina po kojoj se robot kreće klizava (ili recimo na lijevom je dijelu trenutne ćelije bilo blata), lijevi kotač može jedno vrijeme proklizavati i kao posljedica toga robot se može zarotirati u lijevu stranu i na kraju završiti na susjednoj lijevoj ćeliji.

Opisana situacija ilustrirana je na slici 1.2. U trenutku t agent percipira stanje okoline s_t. U tom stanju na raspolaganju mu stoji n akcija koje su na slici označene s $a^{(1)}$ do $a^{(n)}$. Agent bira i izvodi akciju $a_t = a^{(i)}$. Kao rezultat izvođenja te akcije iz stanja s_t okolina može prijeći u jedno od m mogućih stanja koja su na slici označena sa $s^{(1)}$ do $s^{(m)}$ i agentu vratiti pripadnu nagradu (koje su na slici označene s $r^{(1)}$ do $r^{(m)}$). Slika ilustrira situaciju u kojoj je okolina prešla u stanje $s_{t+1} = s^{(j)}$ i pri tome agentu vratila nagradu iznosa $r^{(j)}$. Nakon ovoga, sve bi se ponavljalo: iz novog stanja agent bi odabrao jednu od akcija koje mu stoje na raspolaganju, a okolina bi na temelju toga prešla u jedno od stanja koja su moguća i vratila pripadnu nagradu.

Način na koji agent bira akciju koju želi izvesti nazivamo *politikom* (engl. *policy*). Programski kod koji sučeljem `Policy` modelira politiku prikazan je u nastavku.

Pseudokod 1.2 — Politika: mehanizam odabira akcije.

```
package rl.model;

public interface Policy<S,A> {
  A pickAction(S state);
}
```

Ovisno o vrsti okoline, problem koji agent rješava može biti *epizodički* (ako je gotov u konačnom broju koraka, što podrazumijeva da u okolini mora postojati terminalno stanje) ili *kontinuirani* (ako agent akcije može obavljati do u beskonačnost).

Rješavanje zadatka agent započinje u trenutku $t = 0$ kada se okolina nalazi u stanju s_0. Agent tada bira akciju a_0 čime dobiva nagradu r_1 i okolina prelazi u stanje s_1. Iz tog stanja agent bira akciju a_1 čime dobiva nagradu r_2 i okolina prelazi u stanje s_2. Uz pretpostavku da je zadatak epizodički, ovaj se postupak ponavlja sve do trenutka $t = T$ u kojem okolina prelazi u terminalno stanje s_T, a agent dobiva posljednju nagradu r_T.

Označimo s R_t korigiranu sumu svih nagrada koje je agent primio izvođenjem akcija od trenutka t. Tu ćemo sumu računati prema izrazu (1.1):

$$R_t = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \gamma^3 \cdot r_{t+4} + \cdots + \gamma^{T-t-1} \cdot r_T = \sum_{k=0}^{T-t-1} \gamma^k \cdot r_{t+k+1} \tag{1.1}$$

pri čemu nam faktor $\gamma \in [0,1]$ omogućava da definiramo koliko nam je važna nagrada koju dobivamo kao odgovor na akciju koju radimo u trenutnom stanju u odnosu na nagrade koje ćemo dobiti nakon toga do ostatka epizode. Ovu interpretaciju lagano je vidjeti ako samo malo preradimo prethodni izraz:

$$\begin{aligned} R_t &= r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + \gamma^3 \cdot r_{t+4} + \cdots + \gamma^{T-t-1} \cdot r_T \\ &= r_{t+1} + \gamma \cdot \left(r_{t+2} + \gamma^1 \cdot r_{t+3} + \gamma^2 \cdot r_{t+4} + \cdots + \gamma^{T-t-2} \cdot r_T \right) \\ &= r_{t+1} + \gamma \cdot R_{t+1} \end{aligned}$$

Odabirom $\gamma = 1$ imamo situaciju u kojoj agent direktno pribraja sve nagrade koje mu okolina da do kraja epizode.

Primijetimo da izraz (1.1) vrijedi i za kontinuirane probleme kada je $T = \infty$. U takvom slučaju, da bi korigirana nagrada bila različita od ∞ nužno je koristiti korekcijski faktor $\gamma < 1$.

Također, ako želimo izbjeći različit matematički tretman epizodičkih odnosno kontinuiranih problema, svaki epizodički problem možemo prevesti u kontinuirani na način da sva terminalna

stanja zamijenimo apsorpcijskim stanjima: stanjima u kojima sve akcije vode ponovno u to stanje i sve nagrade koje okolina za akcije izvedene u tom stanju su uvijek nula. Time agent, jednom kada dođe u takvo stanje, kako ono više nije terminalno, nastavlja dalje birati akcije, ali sve ga vraćaju u to isto stanje i nagrade su uvijek nula. Stoga će korigirana suma koja sada ima beskonačno članova i dalje biti jednaka korigiranoj sumi koju smo imali u epizodičkom slučaju, jer će se nagrade do trenutka $t = T$ poklapati, a za $t > T$ će biti 0 čime neće mijenjati sumu.

Imamo li na raspolaganju neku diskretnu okolinu, politiku prema kojoj agent bira akciju te informaciju o željenom početnom stanju, vrlo jednostavno možemo simulirati interakciju agenta s okolinom sve dok zadatak nije riješen. Programski kod u nastavku prikazuje prikladnu metodu.

Pseudokod 1.3 — Izvođenje jedne epizode.

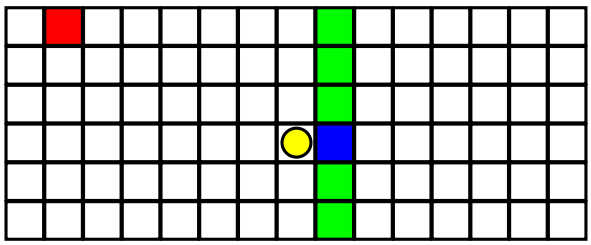
```
public static <S,A> double play(DiscreteEnvironment<S, A> world,
    Policy<S, A> policy, S startState, double gamma) {

  world.setCurrentState(startState);
  double totalReward = 0;
  double scaler = 1;

  while(!world.isFinished()) {
    S currentState = world.getCurrentState();
    A selectedAction = policy.pickAction(currentState);
    double reward = world.applyAction(selectedAction);
    totalReward += scaler * reward;
    scaler *= gamma;
  }

  return totalReward;
}
```

Agenti koje ćemo razvijati imat će zadatak maksimiziranja korigirane sume.

Spomenimo još jedno ograničenje koje ćemo nametnuti okolinama u kojima će agent djelovati: razmatrat ćemo samo okoline koje možemo modelirati Markovljevim procesom odlučivanja. Što to konkretno znači? Razmotrite jednog agenta koji obavlja interakciju s okolinom. Agent je započeo interakciju u trenutku 0, obavio je niz interakcija s okolinom i sada je u vremenskom trenutku t, u stanju s_t, i odabrao je da će izvesti akciju a_t. Okolina kao odgovor na tu akciju s određenim vjerojatnostima može prijeći u neko od sljedećih stanja. Razmotrimo jedno konkretno sljedeće stanje s_{t+1}. Vjerojatnost da će okolina prijeći u to stanje općenito je uvjetna vjerojatnost sljedećeg oblika:

$$p(s_{t+1}|s_t, a_t, s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \ldots, s_0, a_0)$$

Kako bismo ovo malo konkretizirali, pogledajte sliku u nastavku.



Slika prikazuje rešetkasti svijet. Agent je prikazan žutim kružićem i nalazi se pred vratima (plava ćelija); zelenom bojom je prikazan zid - ćelije preko kojih agent ne može prijeći. Kako bi obavio svoj posao, agent treba stići do najdesnije ćelije u prikazanom svijetu. Stigao je pred vrata i pokreće akciju "desno". Što će se dogoditi?

Pretpostavite sada da se u okolini nalazi jedna ćelija sa senzorom pritiska. Ako agent stane na tu ćeliju, aktivira se dozvola otvaranja plavih vrata. Ako ta dozvola nije aktivirana, vrata se neće otvoriti pa kad agent dođe do njih, neće moći proći u desni dio svijeta.

Moramo se još dogovoriti što ćemo smatrati stanjem svijeta. Pretpostavimo da stanjem svijeta smatramo koordinate ćelije na kojoj se nalazi agent, u obliku (redak,stupac); gornja lijeva ćelija tada ima koordinate (0,0), ćelija ispod nje koordinate (1,0). Agent, kako je prikazano na slici, nalazi se na ćeliji (3,7). Zanima nas ako je agent na toj ćeliji i odabrao je akciju "desno", kolika je vjerojatnost da će novo stanje agenta biti (3,8) - drugim riječima, da je proći kroz vrata:

$$p(s_{t+1} = (3,8) | s_t = (3,7), a_t = \text{desno}, s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \ldots, s_0, a_0).$$

Iz danog opisa svijeta sada možemo odgovoriti: ta je vjerojatnost 1 ako je neko od prethodnih stanja s_{t-1} do s_0 bilo stanje $(0,1)$; u suprotnom, ta je vjerojatnost 0. Ukoliko razmatrana uvjetna vjerojatnost ovisi o povijesti izmjene stanja i/ili odabranih akcija, tada za takav svijet kažemo da nije opisiv Markovljevim procesom odlučivanja. *Takve okoline nećemo razmatrati.*

Da bi okolina zadovoljavala svojstvo Markovljevog procesa odlučivanja, uvjetna vjerojatnost mora biti odrediva samo na temelju podataka dostupnih u trenutnom stanju. Stoga će u takvim okolinama vrijediti:

$$p(s_{t+1} | s_t, a_t, s_{t-1}, a_{t-1}, s_{t-2}, a_{t-2}, \ldots, s_0, a_0) = p(s_{t+1} | s_t, a_t)$$

Takve okoline ćemo razmatrati u nastavku.

Kakvu posljedicu ovo ima na način definiranja stanja okoline? Da bi okolina zadovoljavala Markovljevo svojstvo odlučivanja, nužno je da pri definiranju stanja u obzir uzmemo sve relevantne informacije. U opisanom primjeru okolinu bismo mogli uskladiti s Markovljevim procesom odlučivanja tako da stanje definiramo kao uređenu trojku: (redak, stupac, dozvolaAktivirana): prva komponenta stanja predstavlja redak u kojem se agent nalazi, druga stupac u kojem se agent nalazi, a treća je booleova zastavica koja kaže je li agent tijekom života pregazio crvenu ćeliju ili nije.

Primjetite: uz prvotnu definiciju stanja, veličina prostora stanja (broj različitih stanja) bila je $6 \times 15 = 90$. Uz novu definiciju stanja veličina prostora stanja je porasla i iznosi $6 \times 15 \times 2 = 180$. Međutim, uz ovakvu definiciju stanja okolina ima svojstvo Markovljevog procesa odlučivanja. Naime, vrijedi:

1. $p(s_{t+1} = (3,8,da) | s_t = (3,7,da), a_t = \text{desno}) = 1$
2. $p(s_{t+1} = (3,8,ne) | s_t = (3,7,ne), a_t = \text{desno}) = 0$
3. $p(s_{t+1} = (3,8,da) | s_t = (3,7,ne), a_t = \text{desno}) = 0$
4. $p(s_{t+1} = (3,8,ne) | s_t = (3,7,da), a_t = \text{desno}) = 0$

i za odrediti te vjerojatnosti ne trebamo znati koja su bila prethodna stanja niti koje je prethodne akcije agent aktivirao.

Prvi izraz kaže da je vjerojatnost da iz (3,7) uz aktiviranu dozvolu agent otiđe na (3,7) jednaka 1, i pri tome se status dozvole ne mijenja. Drugi izraz kaže da ako agent nema dozvolu, neće moći proći kroz vrata. Treći i četvrti izraz također dolaze iz svojstava opisane okoline. Treći izraz kaže da je nemoguće da je agent bio pred vratima bez dozvole, te da je akcijom "desno" prošao kroz vrata i magično dobio dozvolu. Četvrti izraz opisuje analognu situaciju.

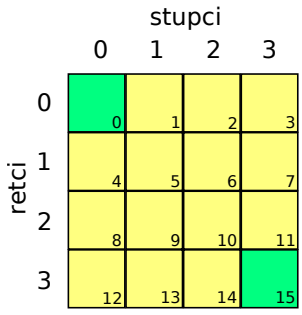
2. Algoritmi temeljeni na modelu okoline

U prethodnom poglavlju opisali smo kontekst u kojem agent djeluje. Pogledajmo sada konkretne primjere okolina i agenata.

Razmatranje ćemo započeti s rešetkastim svijetom dimenzija 4×4 koji se sastoji od 16 ćelija i prikazan je na slici 2.1. Ćelije u svijetu možemo adresirati navođenjem retka i stupca u kojem se nalaze (pa bi tako "koordinate" ćelija išle od (0,0) do (3,3)) ili pak navođenjem rednog broja ćelije (redni brojevi prikazani su u donjem desnom uglu ćelije). Stanjem okoline zvat ćemo informaciju o ćeliji na kojoj se agent trenutno nalazi. Primijetite, to može biti jedan cijeli broj, uređen par brojeva (redak, stupac) i slično. Dvije od ćelija (zelene, ćelije broj 0 i 15) predstavljaju terminalna stanja.

U ovom svijetu, agent može raditi jednu od četiri akcije u svakom od neterminalnih stanja: gore, dolje, lijevo te desno. Pri tome, ako pokuša izvesti akciju kojom bi "ispao" iz svijeta, ostat će na ćeliji na kojoj je bio (primjerice, izvođenjem akcije "gore" dok je agent na ćeliji broj 1 agent će ostati na toj ćeliji); u suprotnom, agent će sigurno otići na ćeliju na koju ga vodi odabrana akcija. Pretpostavimo da izvođenjem svake akcije agent troši jednu jedinicu goriva što mu okolina dojavljuje vraćanjem nagrade iznosa -1.

Primijetimo da nam je prethodni opis specificirao *model okoline*: znamo koja stanja postoje, znamo koje akcije postoje, znamo da je djelovanje akcije deterministično, za svako stanje i odabranu



Slika 2.1: Rešetkasti svijet 1

akciju znamo koje će biti sljedeće stanje te koju ćemo nagradu dobiti. Programski model diskretne okoline ostvaren je sučeljem `DiscreteEnvironmentModel<S,A>` koje je prikazano u nastavku.

Sučelje se sastoji niza metoda koje grupirati u nekoliko cjelina. Postoji metoda kojom možemo dohvatiti faktor korekcije γ koji agent treba koristiti prilikom izračuna korigirane sume. Slijede metode kojima se mogu dohvatiti: skup postojećih stanja, skup postojećih akcija, skup akcija koje su moguće u predanom stanju, skup mogućih prijelaza iz predanog stanja uz predanu akciju te informacija je li predano stanje terminalno. Isključivo kako bi se olakšala izrada algoritama koje ćemo opisati, model podrazumijeva da je moguće napraviti bijektivno preslikavanje između stanja i rednih brojeva odnosno akcija i rednih brojeva (uz pretpostavku da redni brojevi kreću od 0), pa nudi po tri metode: jednu za dohvat rednog broja predanog stanja odnosno akcije, jednu za dohvat stanja odnosno akcije na temelju predanog rednog broja, te jednu koja vraća najveći redni broj koji ima neko stanje odnosno akcija. Ovo će nam omogućiti da stanja i akcije modeliramo složenim objektima (primjerice, kod igre križić-kružić trebali bismo pamtiti tko je stavio što na koju ćeliju), a da istovremeno možemo koristiti polja koja "indeksiramo" po stanjima (odnosno njihovim rednim brojevima) umjesto mapa za učinkovito pamćenje nekih podataka. Konačno, sučelje nudi i pretpostavljenu metodu `createEnvironment` koja stvara i vraća diskretnu okolinu koja se ponaša u skladu s ovim modelom.

Pseudokod 2.1 — Model diskretne okoline.

```
package rl.model;

import java.util.Set;

public interface DiscreteEnvironmentModel<S,A> {

  // Gama
  double getGamma();

  // Dohvat svih postojecih stanja i akcija
  Set<S> getAllStates();
  Set<A> getAllActions();

  // Dohvat akcija mogucih u zadanom stanju
  Set<A> getAvailableActions(S state);

  // Dohvat tranzicija iz predanog stanja uz zadanu akciju
  Set<TransitionInfo<S>> getProbAndReward(S state, A action);

  // Provjera je li stanje terminalno
  boolean isTerminal(S state);

  // Konverzija stanje <-> redni broj
  int getStateIndex(S state);
  S getStateForIndex(int index);
  int getHighestStateIndex();

  // Konverzija akcija <-> redni broj
  int getActionIndex(A action);
  A getActionForIndex(int index);
  int getHighestActionIndex();
```

```
  // Stvori ovakvu okolinu
  default DiscreteEnvironment<S,A> createEnvironment() {...}
}
```

Metoda `getProbAndReward` vraća skup objekata tipa `TransitionInfo<S>`. Ovi objekti omogućavaju pamćenje tri podatka: u koje stanje se prelazi, koja je vjerojatnost da ćemo slijediti ovaj prijelaz te koliku ćemo nagradu dobiti ako ga slijedimo.

Pseudokod 2.2 — Jedan prijelaz.

```
package rl.model;

public class TransitionInfo<S> {
  private S state;
  private double probability;
  private double reward;

  public TransitionInfo(S state, double probability, double reward)
      {
    super();
    this.state = state;
    this.probability = probability;
    this.reward = reward;
  }

  public S getState() {
    return state;
  }

  public double getProbability() {
    return probability;
  }

  public double getReward() {
    return reward;
  }
}
```

Na web-sjedištu s kojeg ste dohvatili ovaj dokument nalazi se i ZIP-arhiva s projektom u kojem su dostupni svi programski kodovi, pa razrede čiji kôd nećemo prikazati u nastavku (ili ne sav kôd) možete detaljnije proučiti u projektu.

A prvi takav razred je pomoćni apstraktni razred `AbstractDiscreteEnvironmentModel<S, A>`. Taj razred implementira sučelje `DiscreteEnvironmentModel<S,A>`, i omogućava pamćenje svih relevantnih podataka u nekoliko mapa i skupova. Također nudi dvije zaštićene metode koje će konkretni razredi moći iskoristiti za jednostavniju izgradnju modela. Prva takva je metoda `tr(S state, double probability, double reward)` koja stvara i vraća primjerak razreda `TransitionInfo<S>`, a druga je `register(S state, A action, TransitionInfo<S>... trs)` koja prima stanje i akciju te varijabilni broj mogućih prijelaza i pamti ih.

Uz pomoć navedenog razreda, sada smo spremni prikazati razred koji predstavlja implementaciju modela opisanog rešetkastog svijeta. Evo relevantnih kodova u nastavku. Prvi kôd je enum koji modelira akciju, drugi kôd je razred koji predstavlja implementaciju modela svijeta. Stanja su modelirana tipom `Integer` (odnosno izravno koristimo redne brojeve ćelija kako je prikazano na

slici 2.1).

Pseudokod 2.3 — Model akcije.

```
package rl.example01;

public enum Action {
  UP, DOWN, LEFT, RIGHT;
}
```

Pseudokod 2.4 — Model rešetkastog svijeta.

```
package rl.example01;

import java.util.Comparator;
import rl.model.environment.AbstractDiscreteEnvironmentModel;

class GridWorldModel extends AbstractDiscreteEnvironmentModel<
    Integer, Action> {
  private Action[] allActions = Action.values();
  private int rows = 4;
  private int cols = 4;

  public GridWorldModel(double gamma) {
    super(gamma, Comparator.naturalOrder(), Comparator.naturalOrder
        ());

    for(int row = 0; row < rows; row++) {
      for(int col = 0; col < cols; col++) {
        if((row==0 && col==0) || (row==3 && col==3)) continue;

        int stateIndex = row*cols + col;

        register(stateIndex, Action.LEFT,
          tr(col>0 ? toStateIndex(row, col-1) : stateIndex, 1, -1)
        );
        register(stateIndex, Action.RIGHT,
          tr(col<cols-1 ? toStateIndex(row, col+1) : stateIndex, 1,
              -1)
        );
        register(stateIndex, Action.UP,
          tr(row>0 ? toStateIndex(row-1, col) : stateIndex, 1, -1)
        );
        register(stateIndex, Action.DOWN,
          tr(row<rows-1 ? toStateIndex(row+1, col) : stateIndex, 1,
              -1)
        );
      }
    }
  }

  private int toStateIndex(int row, int col) {
```

```
    return row*cols + col;
  }

  @Override
  public boolean isTerminal(Integer state) {
    return state==0 || state==15;
  }

  @Override
  public int getStateIndex(Integer state) {
    return state;
  }

  @Override
  public Integer getStateForIndex(int index) {
    return index;
  }

  @Override
  public int getActionIndex(Action action) {
    return action.ordinal();
  }

  @Override
  public Action getActionForIndex(int index) {
    return allActions[index];
  }

}
```

Pogledajte konstruktor razreda `GridWorldModel`. Prolazi se po svim retcima i stupcima. Ako se radi o ćeliji (0,0) ili (3,3), one se preskaču. U suprotnom, za svaku od četiri moguće akcije registrira se točno jedan prijelaz na izračunato sljedeće stanje s vjerojatnošću 1 i uz vraćenu nagradu -1.

Imamo li na raspolaganju model okoline, lagano je napraviti razred koji predstavlja diskretnu okolinu i ponaša se u skladu s tim modelom. Prikladna implementaciju za to već nudi samo sučelje `DiscreteEnvironmentModel<S,A>`, kroz metodu `createEnvironment` (izvorni kôd dostupan je u projektu). Objekt koji ćemo dobiti pozivom te metode pamti trenutno stanje, pa kada mu agent pošalje akciju a, pita model za sve prijelaze koji su akcijom a mogući iz trenutnog stanja, te u skladu s vraćenim vjerojatnostima odabire jedan od prijelaza koji će slijediti, čime dobiva novo trenutno stanje i agentu vraća nagradu zapisanu uz odabrani prijelaz.

Opremljeni ovime, želimo doznati odgovor na sljedeće pitanje: ako agenta početno postavimo na neku ćeliju, koliku će ukupnu nagradu agent dobiti birajući akcije koje ga vode prema terminalnom stanju? Da bismo mogli odgovoriti na to pitanje, moramo znati koju politiku agent koristi za biranje poteza, pa do rješenja možemo doći eksperimentom. Ako je agentova politika deterministička i ako je okolina deterministička, tada možemo iskoristiti prethodno napisanu metodu `play` koja će "odigrati" iz predanog početnog stanja poteze do kraja i vratiti nam ukupnu nagradu. Međutim, ako postoji nedeterminizam (bilo u agentovoj politici, bilo u reakciji okoline), tada nam jedna odigrana epizoda neće predstavljati vjerodostojnu informaciju. Umjesto toga, trebat ćemo eksperiment ponoviti određen broj puta pa izračunati koliko iznosi očekivana nagrada (koju ćemo procijeniti kao aritmetičku sredinu nagrada dobivenih kroz sve epizode). Metoda koja ovo radi prikazana je u nastavku.

Pseudokod 2.5 — Procjena ukupnih dobivenih nagrada.

```
    public static <S,A> double[] estimateStateValues(
    DiscreteEnvironment<S, A> env, DiscreteEnvironmentModel<S, A>
    model, Policy<S, A> policy) {
  final int n = model.getHighestStateIndex() + 1;
  final int REPEATS = 100_000;
  double[] values = new double[n];

  for(S state : model.getAllStates()) {
    int stateIndex = model.getStateIndex(state);
    for(int iteration = 0; iteration < REPEATS; iteration++) {
      values[stateIndex] += Util.play(env, policy, state, model.
          getGamma());
    }
    values[stateIndex] /= REPEATS;
  }

  return values;
}
```

Da bismo mogli napraviti ove eksperimente, prvi korak jest modelirati politiku koju agent koristi. Model politike ostvaren je sučeljem `PolicyModel<S,A>` koje je prikazano u nastavku.

Pseudokod 2.6 — Model politike.

```
package rl.model;

import java.util.Set;

public interface PolicyModel<S, A> {
  public static interface ActionProbability<A> {
    A getAction();
    double getProbability();
  }

  Set<ActionProbability<A>> getActionProbabilitiesForState(S state);

  default Policy<S,A> createPolicy() {...}
}
```

Sučelje definira metodu `getActionProbabilitiesForState` koja prima stanje te vraća skup uređenih parova (akcija,vjerojatnost); ovaj par je modeliran sučeljem `ActionProbability`. Sučelje nudi i pretpostavljenu metodu koja na temelju tog modela generira objekt koji predstavlja politiku (tijelo te metode možete pogledati u projektu).

Pomoćni razred `EquiprobablePolicyModel<S,A>` predstavlja nedeterministički model politike koja u svakom stanju nasumice bira akciju pri čemu su sve akcije jednako vjerojatne. Razred implementira sučelje `PolicyModel<S,A>` te kroz konstruktor prima skup stanja i skup akcija; za svako stanje registrira da su u njemu moguće sve akcije uz jednaku vjerojatnost.

Pogledajmo sada agenta koji koristi upravo spomenutu nedeterminističku politiku kod koje su u svakom neterminalnom stanju sve su akcije jednako vjerojatne. Neka je faktor korekcije $\gamma = 1$. U tom slučaju apsolutna vrijednost sume za početno stanje s odgovarat će broju koraka koje agent

napravi dok ne dođe do terminalnog stanja (prisjetite se, u modeliranom rešetkastom svijetu agent za svaku akciju od okoline dobiva nagradu iznosa -1). Razred `rl.example01.MainPlaying` predstavlja cjelovit program koji obavlja opisane eksperimente.

Pseudokod 2.7 — Procjena dobivenih nagrada uzorkovanjem.

```
package rl.example01;

import java.util.stream.Collectors;

import rl.model.Policy;
import rl.model.PolicyModel;
import rl.model.environment.ModelBasedDiscreteEnvironment;
import rl.model.policy.EquiprobablePolicyModel;
import rl.util.Util;

public class MainPlaying {

  public static void main(String[] args) {
    GridWorldModel model = new GridWorldModel(1);
    ModelBasedDiscreteEnvironment<Integer, Action> world =
      new ModelBasedDiscreteEnvironment<>(model);

    PolicyModel<Integer, Action> policyModel =
      new EquiprobablePolicyModel<>(
        model.getAllStates().stream()
             .filter(s -> !model.isTerminal(s))
             .collect(Collectors.toSet()),
        model.getAllActions()
    );
    Policy<Integer, Action> policy = policyModel.createPolicy();

    double[] values = Util.estimateStateValues(world, model, policy)
       ;

    Util.printArray("Vrijednosti stanja:", values, 4, 4, true);
  }
}
```

Rezultat izvođenja programa prikazan je u nastavku, uz napomenu da će se pri svakom pokretanju brojke ponešto razlikovati.

```
Vrijednosti stanja:
   0.000  -13.958  -20.076  -21.990
 -14.002  -17.986  -19.875  -20.009
 -20.136  -19.973  -17.992  -14.000
 -22.142  -19.971  -13.889    0.000
```

Pokrenete li puno puta program, primjetit ćete da se sume dobivenih nagrada za svako od stanja grupiraju oko nagrada prikazanih u nastavku.

```
   0  -14  -20  -22
 -14  -18  -20  -20
 -20  -20  -18  -14
 -22  -20  -14    0
```

Primjerice, krene li agent iz ćelije (redak=0,stupac=1), u prosjeku će trebati 14 koraka do terminalnog stanja, odnosno do kraja epizode. Iz ćelije (redak=0,stupac=2) u projeku će trebati 20 koraka a iz ćelije (redak=0,stupac=3) čak 22 koraka do terminalnog stanja.

Hoće li se rezultat promijeniti ako promijenimo politiku koju koristi agent? Odgovor je naravno potvrdan. Primjerice, neka agent koristi nedeterminističku politiku pri čemu u 70% slučajeva bira akciju "gore", a preostale tri akcije u jednaku vjerojatnost (svaku uz 10%).

Razred `rl.example01.MainPlaying2` predstavlja cjelovit program koji obavlja eksperimente uz ovu novu politiku agenta. Programski kôd kao i razred koji modelira politiku dostupan je u projektu. Pokretanjem, dobit ćemo sljedeći rezultat (ili nešto slično).

```
Vrijednosti stanja:
   0.000  -29.911  -48.873  -58.008
  -5.658  -30.672  -48.666  -57.477
 -10.724  -31.219  -47.268  -51.167
 -14.545  -31.685  -41.595    0.000
```

Očekivano, prikazana matrica nije više simetrična. Kako sada agent preferira akciju "gore", očekivana suma nagrada iz ćelije (redak=1,stupac=0) je manjeg apsolutnog iznosa usporedimo li je s ćelijom (redak=0,stupac=1) iako su obje susjedi terminalnog stanja. Naime, kako agent preferira akciju "gore", iz ćelije (redak=1,stupac=0) će često već u jednom koraku ući u terminalno stanje i dobiti samo -1 kao ukupnu nagradu, a rjeđe će otići u neku susjednu ćeliju pa od tamo ponovno dalje vrludati. Iz ćelije (redak=0,stupac=1) uz istu preferiranu akciju agent će se često bezuspješno pokušavati pomaknuti prema gore (i dobivati nagrade iznosa -1) pa uz ostala vrludanja koja će raditi, trebat će mu puno više poteza prije no što dođe u terminalno stanje.

Isprobajte

Ova dva primjera možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example01.MainPlaying`
`rl.example01.MainPlaying2`

Ako ste pokrenuli na vlastitom računalu ove eksperimente, vjerojatno ste primjetili da je program radio jedno vrijeme prije no što je ispisao rezultat. Ovakav način procjene koji se temelji na uzorkovanju spor je iz dva razloga: za svako stanje, eksperiment treba ponavljati puno puta, a svaka epizoda pri tome može trajati poprilično vremena (jer agent nasumice bira poteze pa se može vrtiti i u krug jedno vrijeme). Dodatno, procjene na ovaj način možemo napraviti samo ako svako stanje možemo postaviti kao početno i zatim iz njega igrati epizodu. U praksi ovo često nije moguće: razvijate li automatskog igrača neke igre, početno stanje će uvijek biti isto: ono iz kojeg igra počinje (zamislite da razvijate agenta koji upravlja igračem igre Doom).

Na ovom mjestu uvest ćemo dvije nove (i važne) oznake. $v_\pi(s)$ će predstavljati očekivani iznos korigirane sume svih primljenih nagrada počev od trenutka t u kojem se nalazimo u stanju s i agent slijedi politiku π. Naime, iz prethodna dva razmatranja trebalo bi biti jasno da očekivani iznosi ovise o politici koju agent slijedi. $v_\pi(s)$ ćemo također zvati i *vrijednošću* stanja. Formalno možemo

zapisati:

$$v_\pi(s) = \mathbb{E}_\pi[R_t|s_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{k+t+1}|s_t = s\right] \tag{2.1}$$

Funkciju $v_\pi()$ zvat ćemo *funkcijom vrijednosti* pod politikom π (engl. *value function for policy* π).

$q_\pi(s,a)$ će predstavljati očekivani iznos korigirane sume svih primljenih nagrada počev od trenutka t u kojem se nalazimo u stanju s i ako taj prvi puta (dakle u trenutku t) agent odabere akciju a, a nakon toga dalje slijedi politiku π (što znači da ako se agent kasnije tijekom epizode ponovno nađe u ovom istom stanju s, tada će odabrati akciju koju mu diktira njegova politika π, a ne nužno akciju a). Formalno možemo pisati:

$$q_\pi(s,a) = \mathbb{E}_\pi[R_t|s_t = s, a_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{k+t+1}|s_t = s, a_t = a\right] \tag{2.2}$$

Funkciju $q_\pi()$ zvat ćemo *funkcijom vrijednosti akcije* pod politikom π (engl. *action-value function for policy* π).

Izraz za funkciju vrijednosti možemo malo preraditi tako da prikažemo vezu između vrijednosti trenutnog i sljedećeg stanja. Evo kako.

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{k+t+1}\middle| s_t = s\right] \\
&= \mathbb{E}_\pi\left[r_{t+1} + \sum_{k=1}^{\infty} \gamma^k r_{k+t+1}\middle| s_t = s\right] \\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma\cdot\sum_{k=1}^{\infty} \gamma^{k-1} r_{k+t+1}\middle| s_t = s\right] \\
&= \mathbb{E}_\pi\left[r_{t+1} + \gamma\cdot\sum_{k=0}^{\infty} \gamma^{k} r_{k+t+2}\middle| s_t = s\right]
\end{aligned}$$

Kako odrediti očekivani iznos ove sume? Primijetite što znamo: nalazimo se u stanju s. Agent u općem slučaju koristi nedeterminističku politiku. Označimo s $\pi(a|s)$ vjerojatnost da agent u stanju s odabere akciju a. Ako agent odabere akciju a i njome djeluje na okolinu, okolina može biti nedeterministička. Označit ćemo stoga s $p(s'|s,a)$ vjerojatnost da će okolina iz stanja s pod utjecajem akcije a prijeći u stanje s'. Ako okolina prijeđe u stanje s', okolina će agentu vratiti nagradu $r(s,a,s')$ i to je u tom konkretnom slučaju naš r_{t+1}, pa njemu dalje moramo pribrojiti korigirane nagrade koje ćemo dobiti igrajući dalje iz stanja s'.

Ovo dalje znači da moramo proći po svim akcijama i s vjerojatnosti odabira akcije pomnožiti nagradu koju nam vraća okolina. No kako je ista nedetrimistička, trebamo proći po svim mogućim stanjima u koje okolina može prijeći pod akcijom a, te s vjerojatnošću tog prijelaza pomnožiti izravno dobivenu nagradu uvećanu za korigirane nagrade daljnjeg igranja. Evo konkretnog izraza:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma\cdot\mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k r_{k+t+2}\middle| s_{t+1} = s'\right]\right]$$

Konačno, uočimo li da je preostalo očekivanje zapravo jednako iznosu funkcije vrijednosti stanja s', možemo pisati:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\left[r(s,a,s') + \gamma\cdot v_\pi(s')\right] \tag{2.3}$$

odnosno ako uočimo da unutarnja suma zapravo predstavlja q-vrijednosti:

$$v_\pi(s) = \sum_a \pi(a|s) \cdot q_\pi(s,a) \tag{2.4}$$

Time smo konačno spremni za prvo interesantno pitanje: možemo li u našem rešetkastom svijetu odrediti očekivane iznose suma za sve ćelije, što sada znamo da su zapravo vrijednosti $v_\pi(s)$, na neki učinkovitiji način od algoritma uzorkovanja koji smo početno implementirali? **Ako na raspolaganju imamo model okoline i model politike**, tada je odgovor potvrdan i zove se *vrednovanje politike* (engl. *policy evaluation*).

Pogledajte ponovno izraz (2.3) koji smo malo prije izveli. Funkcija $\pi(a|s)$ u našoj je programskoj implementaciji modelirana sučeljem `PolicyModel`. Podatke $p(s'|s,a)$ i $r(s,a,s')$ (drugim riječima, ako smo u stanju s i poduzmemo akciju a, kolika je vjerojatnost da ćemo prijeći u stanje s' i koliko ćemo nagradu dobiti, kao i parametar γ čuva naš model diskretne okoline, sučelje `DiscreteEnvironmentModel`.

Vrednovanje politike je iterativni postupak koji kaže sljedeće. Pretpostavimo da su vrijednosti neterminalnih stanja neke (za svako stanje možemo odabrati neki slučajni broj, a možemo i sve postaviti na nulu). Vrijednosti terminalnih stanja znamo i moramo ih postaviti na 0 (iz njih dalje ne možemo vući poteze niti dobivati ikakve nagrade). Sada za svako stanje na temelju pretpostavljenih vrijednosti i izraza (2.3) izračunaj nove procjene i prati maksimalan iznos za koji se promjenila procjena vrijednosti stanja. U programskoj implementaciji, za ovo bismo koristili dva polja. Prvo polje bi čuvalo procjene vrijednosti stanja u iteraciji i, a nove procjene izračunate na temelju tih vrijednosti bismo pohranili u drugo polje. Jednom kad smo to napravili za sva stanja, polja možemo zamijeniti (ili drugo iskopirati ponovno u prvo), i krenuti u novu iteraciju. Iteracije ponavljamo tako dugo dok najveća promjena vrijednosti stanja ne postane manja od nekog zadanog praga. Možda je na prvi pogled neočekivano, ali postoje garancije da opisani postupak doista konvergira.

U nastavku je dana programska implementacija vrednovanja politike.

Pseudokod 2.8 — Vrednovanje politike.

```
public static <S,A> double[] policyEvaluation(
    DiscreteEnvironmentModel<S, A> model, PolicyModel<S, A>
    policyModel, double threshold) {
  int n = model.getHighestStateIndex() + 1;
  double[] values = new double[n];
  double[] newvalues = new double[n];

  while(true) {
    double delta = 0;
    for(S state : model.getAllStates()) {
      if(model.isTerminal(state)) continue;
      double totalSum = 0;
      for(PolicyModel.ActionProbability<A> ap : policyModel.
          getActionProbabilitiesForState(state)) {
        double sum = 0;
        for(TransitionInfo<S> tr : model.getProbAndReward(state, ap.
            getAction())) {
          sum += tr.getProbability() * (tr.getReward() + model.
              getGamma()*values[model.getStateIndex(tr.getState())]);
        }
        totalSum += ap.getProbability() * sum;
      }
      int stateIndex = model.getStateIndex(state);
```

```
      newvalues[stateIndex] = totalSum;
      delta = Math.max(delta, Math.abs(newvalues[stateIndex]-values[
          stateIndex]));
    }
    System.arraycopy(newvalues, 0, values, 0, values.length);
    if(delta<threshold) {
      break;
    }
  }

  return values;
}
```

Programi koji ilustriraju vrednovanje politike za rešetkasti svijet i dvije prethodno opisane nedeterminističke politike dostupni su kao razredi `rl.example01.MainPolicyEvaluate` i `rl.example01.MainPolicyEvaluate2`. Pokretanjem prvog programa dobit ćemo ispis:

```
Vrijednosti stanja:
   0.000  -14.000  -20.000  -22.000
 -14.000  -18.000  -20.000  -20.000
 -20.000  -20.000  -18.000  -14.000
 -22.000  -20.000  -14.000    0.000
```

a pokretanjem drugog programa:

```
Vrijednosti stanja:
   0.000  -29.784  -48.928  -58.205
  -5.670  -30.423  -48.796  -57.481
 -10.607  -31.280  -47.557  -51.103
 -14.490  -31.675  -41.620    0.000
```

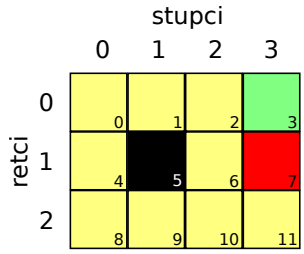
Postupak ispravne vrijednosti određuje brzo i precizno. Primjerice, vrednovanje politike u prvom primjeru zaustavljeno je nakon provedenih 258 iteracija.

Isprobajte

Ova dva primjera možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example01.MainPolicyEvaluate`
`rl.example01.MainPolicyEvaluate2`

Ako se pitate zašto ovaj postupak funkcionira, evo intuitivnog objašnjenja. Pretpostavimo da smo početne procjene vrijednosti svih neterminalnih stanja generirali nasumično. Istina je: u tom se slučaju u prvoj iteraciji procjena nove vrijednost primjerice stanja ćelije (1,1) u cijelosti temelji, kolokvijalno govoreći, na smeću (na trenutnim procjenama vrijednosti stanja (0,1), (2,1), (1,0), (1,2), a te su sve generirane slučajno. No to nije u cijelosti istina za ćelije (0,1), (1,0), (2,3) i (3,2): one istinitu informaciju povlače iz susjednih terminalnih stanja pa nova procjena više nije potpuni šum već je bliža stvarnoj. U sljedećoj iteraciji ponovno radimo procjenu za spomenutu ćeliju (1,1) i primjetite sada: ona dio informacije dobiva od ćelija (0,1) i (1,0) za koje smo upravo konstatirali da sadrže i zrnce istine; time će u ovoj iteraciji procjena vrijednosti ove ćelije postati kvalitetnija. Kako iteracije napreduju, tako će se "istina" malo po malo širiti iz terminalnih stanja prema svim stanjima i postupak će u jednom trenutku konvergirati.

Pogledajmo još jedan primjer rešetkastog svijeta koji je prikazan na slici 2.2.



Slika 2.2: Rešetkasti svijet 2

Za razliku od determinističke okoline koju smo opisali uz sliku 2.1, ova okolina je nedeterministička: kada agent zatraži izvođenje neke akcije, ako ista nije moguća, stanje će ostati nepromijenjeno; međutim, ako je akcija moguća, tada će se promjena stanja u očekivano dogoditi u 80% slučajeva. U preostalih 20% slučajeva, dogodit će se promjena kao da je zatražena jedna od akcija koja nije suprotna traženoj akciji, i to svaka u 10% slučajeva. Konkretno, ako se agent nalazi u ćeliji 10 i na okolinu djeluje akcijom "gore", u 80% slučajeva agent će se naći na ćeliji 6; u 10% slučajeva naći će se na ćeliji 9, a u preostalih 10% slučajeva naći će se na ćeliji 11 (dakle, kao da su bile zatražene akcije "lijevo" ili "desno"; akcija "dolje" suprotna je od odabrane akcije "gore"). Stanja 3 i 7 su terminalna stanja. Ćelija 5 predstavlja zid - agent ne može doći na tu ćeliju. Prelaskom u stanje 3 agent dobiva nagradu iznosa +1. Prelaskom u stanje 7 agent dobiva nagradu iznosa -1. Za sve ostale izvedene akcije iz bilo kojeg stanja agent uvijek dobiva nagradu iznosa r (gdje će r biti parametar, kako bismo mogli istražiti ponašanja uz različite iznose te nagrade). Kako se ona nagrada dodjeljuje za svaku izvedenu akciju (osim onih kojima prelazimo u terminalna stanja), naziva se nagrada življenje (engl. *living reward*) i tipično je negativnog iznosa. Kako ćemo razmatrati agente koji žele maksimizirati korigiranu sumu nagrada, dobivanje negativnih nagrada će penalizirati agenta čime će on pokušavati razviti takvu politiku kojom će dobiti minimalni broj takvih nagrada (drugim riječima, pokušat će riješiti zadatak što je brže moguće).

Programski model ovakvog svijeta dostupan je u paketu `rl.example02` kroz razrede `Action` i `GridWorldModel`. Ako se agent kreće po ovom svijetu koristeći nasumičan odabir akcija pri čemu su sve akcije jednako vjerojatne, koliki su iznosi funkcije vrijednosti $v_\pi(s)$? Procjenu možemo ponovno napraviti uzorkovanjem (program `MainPlaying`) ili vrednovanjem politike (program `MainPolicyEvaluate`). Rezultat dobiven postupkom vrednovanja politike uz $r = 0$ (dakle, agent se ne kažnjava za duljinu epizode) i faktor korekcije $\gamma = 1$ dan je u nastavku.

```
Vrijednosti stanja:
  -0.038    0.089    0.215    0.000
  -0.165    0.000   -0.443    0.000
  -0.291   -0.418   -0.544   -0.772
```

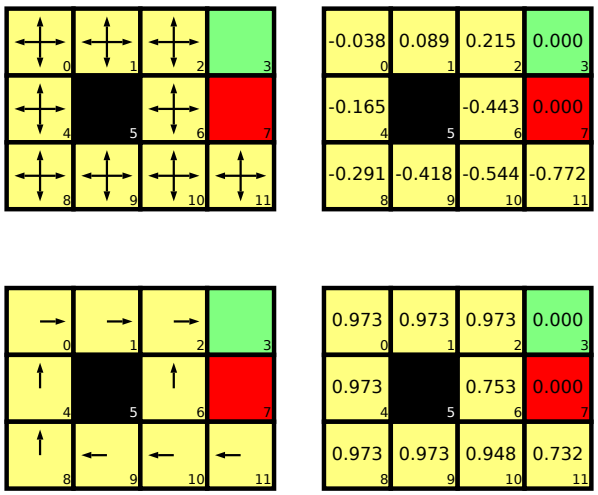
Kako možemo vidjeti, agent će gotovo iz svih stanja završiti s negativnom nagradom; iznimka su stanja (0,1) i (0,2) koja su dovoljno blizu terminalnom stanju (0,3) koje daje nagradu +1 da agent često uspjeva završiti u tom stanju.

Isprobajte

Određivanje vrijednosti stanja uzorkovanjem odnosno algoritmom vrednovanja politike za opisanu okolinu možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example02.MainPlaying`

`rl.example02.MainPolicyEvaluate`

Promjenom politike koju koristi agent promijenit će se i iznosi vrijednosti stanja. Slika u nastavku prikazuje u gornjem dijelu upravo korištenu politiku (lijevo) i vrijednosti stanja (desno); u donjem dijelu prikazuje drugu politiku (lijevo) i vrijednosti stanja (desno). Ova posljednja politika ručno je konstruirana tako da minimizira broj koraka koje agent treba napraviti da bi došao do terminalnog stanja koje daje nagradu +1.



Primijetite kako uz ovu novu politiku vrijednost stanja (0,2) - ćelije koja je lijevo od terminalnog stanja koje daje nagradu +1 nije 1 već je manja. Agent u ovom scenariju koristi deterministčku politiku koja mu kaže da na toj ćeliji napravi akciju "desno" čime pokušava doći na terminalnu ćeliju i pokupiti nagradu iznosa 1. Međutim, ovu smo okolinu definirali kao nedeterminističku: u 80% slučajeva, to će se dogoditi. U 10% slučajeva okolina će se ponašati kao da je agent zatražio akciju "gore", čime će ostati na ćeliji (0,2). U preostalih 10% slučajeva, agent će završiti na ćeliji (1,2), koja je tik uz terminalnu ćeliju koja daje nagradu -1. Ako u nekoj epizodi agent doista završi na ćeliji (1,2), politika mu kaže da bira akciju "gore" čime se pokušava vratiti na ćeliju (0,2). Međutim, kako je okolina nedeterministička, u 10% slučajeva ta će akcija agenta baciti u terminalno stanje (1,3) uz nagradu -1. I upravo ovo je razlog zašto je vrijednost stanja (0,2) manja od 1.

Isprobajte

Određivanje vrijednosti stanja uzorkovanjem odnosno algoritmom vrednovanja politike za opisano ponašanje agenta možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example02.MainPlaying`
`rl.example02.MainPolicyEvaluate`
ali prije pokretanja predajte programima jedan argument koji će primiti prilikom pokretanja: policy2.

Postavlja se pitanje možemo li odrediti koliko iznose vrijednosti stanja kada bi agent koristio optimalnu politiku, i dakako, možemo li onda rekonstruirati tu optimalnu politiku? Odgovor je ponovno potvrdan.

2.1 Iteracija vrijednosti

Postupak poznat pod nazivom *Iteracija vrijednosti* (engl. *Value Iteration*) je iterativni postupak koji, **uz pretpostavku da imamo na raspolaganju model okoline**, može odrediti maksimalne vrijednosti za svako stanje i te vrijednosti tada odgovaraju maksimalnom iznosu korigiranih suma nagrada koje agent teoretski može dobiti kada kreće iz tih stanja.

Krenimo za početak sa sljedećom pretpostavkom: za svako stanje poznata nam je njegova teorijski moguća maksimalna vrijednost. Pretpostavimo sljedeći konkretan scenarij: okolina je deterministička; agent percipira stanje s i na raspolaganju mu stoje 3 akcije: a_1, a_2 i a_3. Svakom od tih akcija okolina će promijeniti stanje u $s^{(1)}$, $s^{(2)}$ odnosno $s^{(3)}$ i vratiti nagradu $r^{(1)}$, $r^{(2)}$ odnosno $r^{(3)}$.

Ako se agent odluči poduzeti akciju a_1, prijeći će dakle u stanje $s^{(1)}$ i od okoline dobiti nagradu $r^{(1)} = r(s,a_1,s^{(1)})$ te sve nagrade koje će dalje dobiti igrajući iz stanja $s^{(1)}$:

$$R^{(a_1)} = r(s,a_1,s^{(1)}) + \gamma \cdot v(s^{(1)})$$

Isto razmatranje možemo napraviti i za preostale akcije, pa imamo:

$$R^{(a_2)} = r(s,a_2,s^{(2)}) + \gamma \cdot v(s^{(2)})$$
$$R^{(a_3)} = r(s,a_3,s^{(3)}) + \gamma \cdot v(s^{(3)})$$
$$R^{(a_4)} = r(s,a_4,s^{(4)}) + \gamma \cdot v(s^{(4)})$$

Podsjetimo se sada pretpostavke s kojom smo krenuli: sve vrijednosti stanja znamo, i one su optimalne (maksimalne moguće). To znači da su izračunate vrijednosti $R^{(a_1)}$, $R^{(a_2)}$, $R^{(a_3)}$ i $R^{(a_4)}$ točne, i govore nam koliko ćemo ukupno nagradu dobiti ako krećemo iz stanja s i odaberemo akciju a_1, a_2, a_3 odnosno a_4. No tada je dalje jednostavno: ako agent želi maksimizirati sumu nagrada koju dobiva, tada mora odabrati onu od analiziranih akcija uz koju je ukupna suma nagrada maksimalna, i to je tada ujedno i vrijednost stanja s. Drugim riječima, u ovom pojednostavljenom slučaju koji smo razmotrili vrijednost stanja s odgovara:

$$v(s) = \max_{a \in \{a_1,a_2,a_3,a_4\}} \left(r(s,a,s') + \gamma \cdot v_\pi(s') \right)$$

U općenitijem slučaju, okolina može biti nedeterministička, pa kao rezultat odabira akcije a uz određene vjerojatnosti možemo prijeći neko iz skupa stanja i dobiti pripadnu nagradu. U tom slučaju vrijedi:

$$v^*(s) = \max_a \left(\sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma \cdot v^*(s') \right] \right) \tag{2.5}$$

odnosno ako uočimo da maksimum zapravo tražimo po q-vrijednostima:

$$v^*(s) = \max_a q^*(s,a) \tag{2.6}$$

pri čemu smo oznakom $v^*(s)$ označili optimalne vrijednosti stanja, a oznakom $q^*(s,a)$ optimalne q-vrijednosti. Ovo su takozvane Bellmanove jednadžbe. Optimalne vrijednosti stanja odnosno q-vrijednosti su formalno definirane ovako:

$$v^*(s) = \max_\pi v_\pi(s)$$
$$q^*(s,a) = \max_\pi q_\pi(s,a)$$

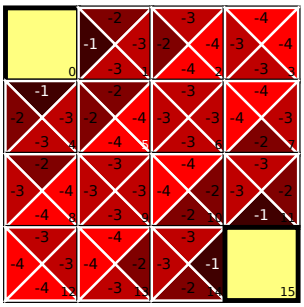
gdje maksimum pretražujemo po svim mogućim politikama. Politiku π za koju se taj maksimum postiže označavat ćemo s π^* i zvati optimalnom politikom.

Iteracija vrijednosti implementacijski je identičan postupak kao vrednovanje politike, samo što za izračun nove procjene vrijednosti stanja ne koristi izraz (2.3) koji smo koristili kod vrednovanja politike, već upravo izvedeni izraz (2.5). Postupak je prikazan u nastavku.

Pseudokod 2.9 — Iteracija vrijednosti.

```
public static <S,A> double[] optimalPolicyValueIteration(
    DiscreteEnvironmentModel<S, A> model, double threshold) {
  int n = model.getHighestStateIndex() + 1;
  double[] values = new double[n];
  double[] newvalues = new double[n];

  while(true) {
    double delta = 0;
    for(S state : model.getAllStates()) {
      if(model.isTerminal(state)) continue;

      double max = Double.NEGATIVE_INFINITY;
      for(A action : model.getAvailableActions(state)) {
        double sum = 0;
        for(TransitionInfo<S> tr : model.getProbAndReward(state,
            action)) {
          sum += tr.getProbability() * (tr.getReward() + model.
              getGamma()*values[model.getStateIndex(tr.getState())]);
        }
        max = Math.max(max, sum);
      }

      int stateIndex = model.getStateIndex(state);
      newvalues[stateIndex] = max;
      delta = Math.max(delta, Math.abs(newvalues[stateIndex]-values[
          stateIndex]));
    }
    System.arraycopy(newvalues, 0, values, 0, values.length);
    if(delta<threshold) {
      break;
    }
  }

  return values;
}
```

Provedemo li iteraciju vrijednosti nad prvim rešetkastim svijetom uz $\gamma = 1$, dobit ćemo rezultat prikazan na slici u nastavku. Lijevi dio slike za svako stanje i akciju prikazuje q-vrijednost; desni dio slike prikazuje vrijednosti stanja.

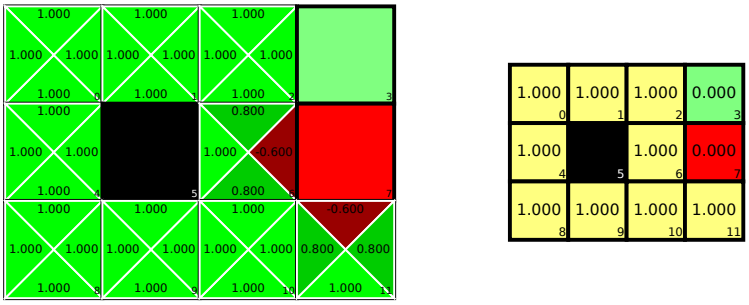


0.00	-1.00	-2.00	-3.00
-1.00	-2.00	-3.00	-2.00
-2.00	-3.00	-2.00	-1.00
-3.00	-2.00	-1.00	0.00

Vidimo da su otkrivene vrijednosti stanja jednake negativnom iznosu minimalnog broja koraka

koji je potreban od stanja do najbližeg terminalnog stanja. Uvjerite se da je vrijednost svakog stanja jednaka maksimalnoj q-vrijednosti tog stanja gledano po svim akcijama. Naš agent koji je koristio nasumičnu politiku postizao je puno gore rezultate od ovdje prikazanih.

Provedemo li iteraciju vrijednosti nad drugim rešetkastim svijetom uz $r = 0$ i $\gamma = 1$ dobit ćemo rezultat prikazan na slici u nastavku. Lijevi dio slike za svako stanje i akciju prikazuje q-vrijednost; desni dio slike prikazuje vrijednosti stanja.



Prethodno smo već napravili analizu vrijednosti stanja ako agent koristi nasumičnu politiku, te ako koristi deterministička politiku koja je djelovala sasvim smisleno. Međutim, čak i uz tu drugu politiku nismo uspijeli niti iz jednog stanja postići da je očekivana suma svih nagrada jednaka 1. Postupkom iteracije vrijednosti otkrili smo pak da, ne samo da s one najbliže ćelije terminalnom stanju možemo osigurati da je suma nagrada jednaka 1, već to možemo ostvariti sa svih stanja! Očito, politika koju smo koristili nije bila optimalna politika.

Isprobajte

Ove primjere možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example01.MainValueIteration`
`rl.example02.MainValueIteration`
pa obratite pažnju na prvi dio ispisa gdje će biti ispisane naučene vrijednosti stanja.

Kako izgledaju optimalne politike koje agent treba koristiti da bi maksimizirao sume primljenih nagrada?

Da bismo odgovorili na prethodno pitanje, morat ćemo se ponovno vratiti na kratku diskusiju koju smo napravili da bismo došli do izraza (2.5). Podsjetimo se što smo tada zaključili: vrijednost stanja odgovara maksimumu suma očekivanih nagrada pogledano po svakoj akciji koju možemo napraviti u stanju s. Optimalna politika π^* tada je ona koja upravo odabire te akcije. Drugim riječima, u stanju s, optimalna politika odabire akciju:

$$\pi^*(s) = \arg\max_a \left(\sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma \cdot v^*(s') \right] \right) \tag{2.7}$$

Uočimo li da su argumenti funkcije max zapravo q-vrijednosti, možemo koristiti i izraz:

$$\pi^*(s) = \arg\max_a \left(q^*(s,a) \right) \tag{2.8}$$

Ovdje korištena funkcija argmax ne vraća maksimalnu od predanih vrijednosti, već vraća argument (a to je u ovom slučaju akcija) za koji se postiže maksimalna vrijednost.

Ako za neko stanje postoji više akcija koje imaju jednak maksimalni iznos, tada bi politika π^* mogla biti i nedeterministička, i s bilo kakvom distribucijom vjerojatnosti u stanju s birati bilo koju od tih akcija koje postižu maksimum.

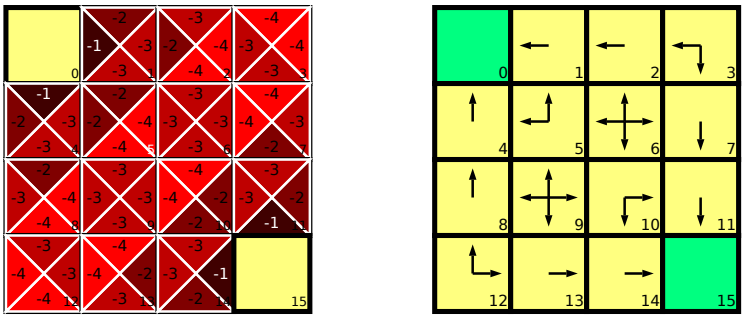
Primijetite da za provođenje ovog postupka trebamo vrijednosti stanja (što smo prethodno odredili iteracijom vrijednosti) te model okoline.

Pseudokod 2.10 — Otkrivanje politike na temelju vrijednosti stanja.

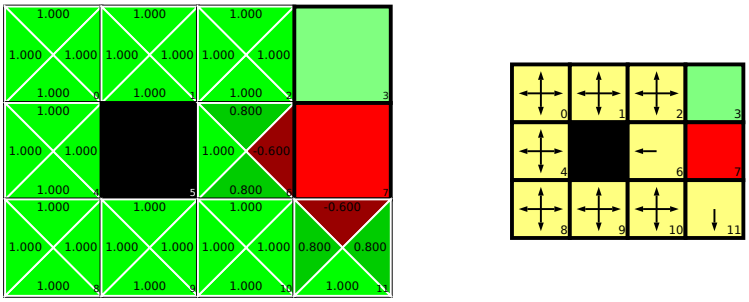
```
public static <S,A> A[][] extractOptimalPolicyFromStateValues(
    DiscreteEnvironmentModel<S, A> model, double[] values, double
    tolerance, IntFunction<A[][]> array2dFactory, IntFunction<A[]>
    array1dFactory) {
  int n = model.getHighestStateIndex()+1;
  A[][] policy = array2dFactory.apply(n);

  List<A> actionList = new ArrayList<>();
  List<Double> valueList = new ArrayList<>();
  for(S state : model.getAllStates()) {
    actionList.clear();
    valueList.clear();
    double max = Double.NEGATIVE_INFINITY;
    for(A action : model.getAvailableActions(state)) {
      double sum = 0;
      for(TransitionInfo<S> tr : model.getProbAndReward(state,
          action)) {
        sum += tr.getProbability() * (tr.getReward() + model.
            getGamma()*values[model.getStateIndex(tr.getState())]);
      }
      actionList.add(action);
      valueList.add(sum);
      if(sum>max) {
        max = sum;
      }
    }
    Set<A> set = new LinkedHashSet<>();
    for(int i = 0; i < actionList.size(); i++) {
      if(Math.abs(max-valueList.get(i))<tolerance) {
        set.add(actionList.get(i));
      }
    }
    policy[model.getStateIndex(state)] = set.toArray(array1dFactory)
        ;
  }

  return policy;
}
```

Prikazani kôd za svako stanje u pomoćnu listu popiše moguće akcije te očekivane dobitke uz te akcije. Potom konstruira skup koji sadrži samo one akcije koje su imale očekivanu vrijednost unutar zadane torancije do maksimalne vrijednosti. Na kraju skup pretvori u polje akcija, i zapamti u polju koje čini politiku. Kod konstruira i vraća polje polja akcija; prvo indeksiranje je po rednom broju stanja, a drugo po akcijama koje u tom stanju imaju maksimalne očekivane sume.

Primijenimo li ovaj postupak na prvi rešetkasti svijet, dobit ćemo rezultat prikazan na slici u nastavku. Lijevo su ponovno prikazane q-vrijednosti svih stanja, a desno optimalna politika. Primijetite: u svakom stanju, optimalna politika dopušta samo one akcije koje u tom stanju imaju najveći iznos q-vrijednosti, gledano po svim akcijama tog stanja.



Primijenimo li postupak na drugi rešetkasti svijet, dobit ćemo rezultat prikazan na slici u nastavku. Lijevo su ponovno prikazane q-vrijednosti svih stanja, a desno optimalna politika.



Za prvi rešetkasti svijet vidimo da je doista otkrivena politika koja agenta usmjerava prema najbližem terminalnom polju. U slučaju da postoji više takvih puteva, sve akcije koje vode po tim putevima su otkrivene. Tako je primjerice za ćeliju (redak=0,stupac=1) optimalna samo jedna akcija: lijevo; za ćeliju (1,1) optimalne su dvije akcije: bilo lijevo, bilo gore; za ćeliju (2,1) dopuštene su sve četiri akcije.

Isprobajte

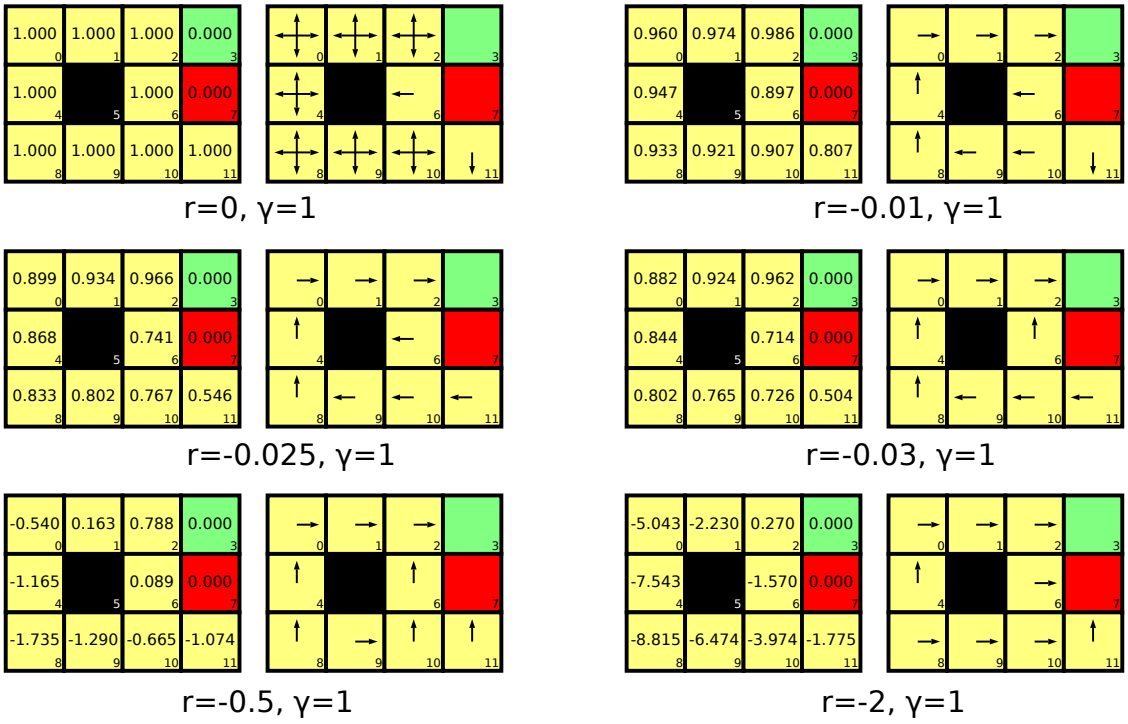
Ove primjere možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example01.MainValueIteration`
`rl.example02.MainValueIteration`
pa obratite pažnju posljednji dio ispisa gdje će biti ispisane optimalne akcije. U središnjem dijelu ispisa bit će prikazana tablica q-vrijednosti.

Rezultat za drugi rešetkasti svijet je zanimljiviji. Kako agenta ne kažnjavamo za broj koraka koje radi tijekom epizode (jer razmatramo slučaj gdje je $r = 0$), optimalna politika za sva neterminalna polja osim (1,2) i (2,3) dopušta sve četiri akcije. Polju (1,2) s desne je strane terminalno polje koje daje nagradu -1; stoga na ovom mjestu optimalna politika propisuje da je jedino dopušteno napraviti akciju lijevo, čime se, iako je okolina nedeterministička, garantira da tom akcijom nećemo završiti na opasnom terminalnom polju. Dakako, u puno pokušaja nećemo se niti maknuti s tog polja (ali kako je $r = 0$, to nas ništa ne košta). Povremeno će nas okolina baciti na gornje ili donje polje i to je OK. Slično vidimo da je otkriveno i za ćeliju (2,3).

Sada možemo pogledati i kako se mijenjaju optimalne politike ovisno o parametrima okoline u kojoj agent djeluje. Na slici u nastavku prikazane su optimalne vrijednosti stanja te optimalne politke koje bi agent trebao koristiti, ovisno o iznosu nagrade r u drugom rešetkastom svijetu.

Prvi slučaj, kad je $r = 0$ već smo prodiskutirali. Preostali slučajevi uvode kaznu za svaki napravljeni korak. U slučaju da je ta kazna vrlo mala ($r = -0.01$), optimalna politika zadržava

oprezno ponašanje na poljima (1,2) i (2,3), dok u svim ostalim agenta usmjerava jedinstvenim putem. Primijetite, taj put agenta koji se zatekne u donjem dijelu svijeta tjera dužim putem prema poželjnom terminalnom stanju (put najprije vodi skroz u lijevo, pa zatim prema gore pa desno do kraja). Razlog što ćeliji (2,2) nije pridružena akcija "gore" jest oprezno rukovanje ćelijom (1,2): na njoj će agent u prosjeku puno puta raditi akciju lijevo, prije no što ga okolina baci ili gore ili dolje, i time će nakupiti dosta negativne nagrade. Stoga je manji trošak krenuti uz donji rub u lijevo pa naokolo.



Uz $r = -0.025$ trošak opreznog izbjegavanja terminalnog stanja s ćelije (2,3) postaje prevelik, pa se politika mijenja tako da agenta šalje lijevo. Daljnjim porastom imamo $r = -0.03$ gdje trošak izbjegavanja terminalnog stanja s ćelije (1,2) postaje prevelik pa se politika mijenja tako da u tom stanju bira akciju gore.

Porastom r na -0.5 agent dolazi u situaciju da mu je s ćelije (2,3) dobro terminalno stanje naprosto predaleko, i na putu do njega akumulirao bi veći iznos negativne nagrade no što će učiniti ako odmah odluči otići u terminalno stanje (1,3); s druge strane, ćelija (1,2) još uvijek nije predaleko. Primijetite i da se sada s donje strane prekida kruženje u lijevo: za ćeliju (2,1), sada je optimalna akcija desno.

Konačno, kad r postane prevelik ($r = 2$) optimalno ponašanje agenta jest da što prije dođe do najbližeg terminalnog stanja, i to lijepo ilustrira prikazana politika.

2.2 Iteracija politike

Iteracija politike alternativni je način izračuna optimalnih vrijednosti stanja te optimalne politike. Prethodno opisani postupak iteracije vrijednosti temelji se na iterativnom ažuriranju stanja uporabom izraza (2.5) koji ponavljamo u nastavku:

$$v^*(s) = \max_a \left(\sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma \cdot v^*(s') \right] \right)$$

Iz izraza je vidljivo da za svako stanje razmatramo svaku akciju na temelju koje računamo ostatak izraza. Složenost jedne iteracije stoga je jednaka umnošku kardinaliteta skupa stanja i kardinaliteta skupa akcija. I taj se postupak zatim ponavlja kroz niz iteracija, sve dok ne dođe do konvergencije.

Nakon toga, na temelju konvergiranih optimalnih vrijednosti stanja određujemo optimalnu politiku oslanjajući se na izraz (2.7) koji također ponavljamo:

$$\pi^*(s) = \arg\max_a \left(\sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma \cdot v^*(s') \right] \right)$$

Složenost jedne iteracije ovog postupka jednaka umnošku kardinaliteta skupa stanja i kardinaliteta skupa akcija. No na sreću, s ovime smo gotovi u jednoj iteraciji.

U situacijama kada je broj stanja te broj mogućih akcija velik, računska složenost opisanih postupaka je velika. Stoga isti posao možemo odraditi na drugi način koji također konvergira ka optimalnoj politici: iteracijom politike.

Iteracija politike je iterativni postupak određivanja optimalne politike koji umjesto iteracije vrijednosti pa otkrivanja politike iterativno koristi vrednovanje politike pa otkrivanje politike (ovaj drugi korak mijenjat će trenutnu politiku pa ćemo ga još zvati ažuriranjem politike). Postupak započinje tako da sami odaberemo jednu konkretnu deterministčku politiku prema kojoj ćemo se ravnati (tu politiku možemo odabrati i nasumično, no ono na što ciljamo je da ta politika svakom od stanja pridruži jednu konkretnu akciju). Uz takvu politiku, označimo je $\pi^{(0)}$ provedemo postupak vrednovanja te politike, koji se u tom slučaju temelji na izrazu 2.3 koji, u slučaju da je politika deterministička, prelazi u:

$$v_{\pi^{(i)}}(s) = \sum_{s'} p(s'|s,\pi^{(i)}(s)) \left[r(s,\pi^{(i)}(s),s') + \gamma \cdot v_{\pi^{(i)}}(s') \right]$$

Primijetite kako sada ne razmatramo za svako stanje svaku akciju: politika nam za *i*-to stanje kaže koju (jednu) akciju agent tada radi, pa iterativni postupak ponavljamo do konvergencije. Složenost ovog postupka tada je proporcionalna samo kardinalitetu skupa stanja. Postupak će konvergirati prema vrijednostima stanja koje možemo očekivati ako agent koristi tu politiku. Primijetimo da to nije optimalna politika, pa niti konvergirane vrijednosti stanja nisu optimalne.

No sada, nakon što je postupak određivanja vrijednosti stanja uz politiku $\pi^{(0)}$ konvergirao, možemo na temelju tih vrijednosti prema izrazu (2.7) odrediti novu politiku. Ovaj postupak će za svako stanje razmotriti sve akcije, i za neko stanje može promijeniti odluku o akciji koju agent treba raditi jer možemo utvrditi da ćemo tom promjenom očekivano dobivati veću nagradu; stoga ovaj korak nazivamo ažuriranjem politike (engl. *policy improvement*). Ako nije bilo promjena u politici, kompletan postupak je gotov i konvergirane vrijednosti stanja su optimalne, kao i politika koju smo koristili.

Ako je bilo promjena u politici, tada imamo novu politiku $\pi^{(1)}$. Za nju ponavljamo oba koraka: radimo vrednovanje te politike (i to možemo tako da samo nastavimo ažurirati prethodno konvergirane vrijednosti koje smo dobili prethodnom politikom), što je ponovno računski manje zahtjevno, pa kad postupak konvergira, ponovno na temelju konvergiranih stanja odredimo novu politku, i tako ponavljamo dok ne dođemo do situacije da nije bilo promjena u politici. Konceptualno, radimo sljedeći niz koraka:

1. Nasumice odredimo početnu politiku $\pi^{(0)}$
2. $v^{(0)}$ odredi vrednovanjem politike $\pi^{(0)}$
3. Na temelju $v^{(0)}$ odredi novu politiku $\pi^{(1)}$ ažuriranjem politike
4. $v^{(1)}$ odredi vrednovanjem politike $\pi^{(1)}$
5. Na temelju $v^{(1)}$ odredi novu politiku $\pi^{(2)}$ ažuriranjem politike
6. $v^{(2)}$ odredi vrednovanjem politike $\pi^{(2)}$
7. Na temelju $v^{(2)}$ odredi novu politiku $\pi^{(3)}$ ažuriranjem politike

8. ...
9. Ponavljaj vrednovanje politike i ažuriranja politike sve dok se politika mijenja.

Ono što dobivamo opisanim postupkom jest da radimo puno "jefitinih" iteracija vrednovanja politike pa jednu "skupu" iteraciju otkrivanja bolje politike, i to zatim ciklički ponavljamo.

Implementacija ovog postupka dostupna je u priloženom projektu gdje možete pogledati programski kôd.

Prilikom implementacije opisanog postupka može se dogoditi rubni slučaj koji treba uzeti u obzir. Prisjetite se što smo rekli - zašto postupak vrednovanja politke konvergira? Razlog je bio taj što "istinite" vrijednosti koje tipično imamo barem uz terminalna stanja malo po malo će nadjačati šum koji smo uveli koristeći nasumične vrijednosti za početne vrijednosti stanja. Ako se dogodi da početna politika koju nasumično generiramo je takva da agent uvijek iz nekog terminalnog stanja ide u neko drugo neterminalno stanje (i to bude istina za sva neterminalna stanja) - tada će se vrijednosti svakog stanja neprestano računati samo ne temelju slučajno odabranih vrijednosti, i taj postupak može ne konvergirati. Kako bi se tome doskočilo, možemo uvesti limit na broj iteracija koji smo premni napraviti. Jednom kad "preživimo" prvo vrednovanje politike, postupak određivanja nove politike za svako stanje razmatra sve akcije i on će tipično politiku promijeniti tako da se iz stanja bliskih "dobrim" terminalnima odaberu prikladne akcije, čime ćemo u sljedećem ciklusu postupak vrednovanja politike uspjeti dovesti do konvergencije. Alternativno, početnu politiku možemo napraviti tako da za svako stanje sve akcije budu jednako vjerojatne - taj će postupak tada konvergirati, ali trebat će određen broj iteracija.

Isprobajte

Iteraciju politike nad oba rešetkasta svijeta možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example01.MainPolicyIteration`
`rl.example02.MainPolicyIteration`
pri čemu na primjeru rešetkastog svijeta 2 u pozivu konstruktora modela možete mijenjati iznos nagrade r pa vidjeti kako ona utječe na naučenu politiku.

2.3 Ponavljanje

Kroz ovo poglavlje upoznali smo se s nekoliko važnih pojmova i algoritama. Stoga ćemo ih ovdje ponoviti.

Razmatrali smo okoline čija su stanja diskretna i akcije diskretne. Okolina je mogla biti deterministička ili nedetrimistička. Radili smo uz pretpostavku da se okolina ponaša u skladu s Markovljevim procesom odlučivanja, odnosno da za donijeti odluku u nekom koraku povijest nema nikakvog utjecaja; bitna je samo trenutna situacija.

Definirali smo pojam *politike* kako bismo opisali način na koji agent, kada percipira neko stanje okoline, bira akciju koju će izvršiti. Agent je mogao slijediti determinističku ili nedeterminističku politiku.

Definirali smo pojam *vrijednost stanja* uz zadanu politiku, kao očekivanu korigiranu sumu dobitaka, odnosno ukupnu nagradu koju agent može očekivati ako krene igrati iz tog stanja i slijedi zadanu politiku.

Definirali smo pojam *q-vrijednost stanja i akcije* uz zadanu politiku (što kraće zovemo samo q-vrijednost), kao očekivanu korigiranu sumu dobitaka, odnosno ukupnu nagradu koju agent može očekivati ako krene igrati iz tog stanja i tada napravi zadanu akciju, a nakon toga dalje slijedi zadanu politiku.

Ako smo imali na raspolaganju okolinu koju možemo simulirati i kojoj smo mogli proizvoljno postavljati početno stanje, pokazali smo (konceptualno najjednostavniji) način na koji možemo

odrediti vrijednosti stanja: statističkim uzorkovanjem. Puno puta bismo agenta postavili u neko stanje, odigrali igru do kraja i računali dobivene nagrade te na kraju odredili srednju vrijednost dobivenih nagrada kroz sve odigrane epizode. Mana ovakvog postupka bila je računska složenost.

Potom smo napravili **jednu jaku pretpostavku**: rekli smo da znamo kako izgleda *model okoline*. Drugim riječima, pretpostavili smo da raspolažemo s informacijama koje nam govore kolike su vjerojatnosti da se pod utjecajem agenta u okolini dogodi određena promjena stanja te koliku će nagradu agent dobiti.

I ono što je sada bitno, samo uz tu pretpostavku pokazali smo da postoji učinkovitiji način:

- kako odrediti vrijednosti stanja ako agent slijedi zadanu politiku:
  $\rightarrow$ postupkom *vrednovanja politike* te
- kako odrediti optimalne vrijednosti stanja:
  $\rightarrow$ postupkom *iteracije vrijednosti*
  $\rightarrow$ postupkom *iteracije politike*.

Od svega što smo do sada naučili, ništa zapravo nisu algoritmi podržanog učenja. Međutim, razumijevanje istih nužno je kako bismo mogli razumjeti problem koji rješava podržano učenje.

3. Podržano učenje

Što je, zapravo, problem koji rješava podržano učenje? Kao i u prethodnom poglavlju, razmatramo okoline u kojima djeluje agent i koje imaju karakteristike Markovljevih procesa odlučivanja. Za okolinu stoga možemo definirati skup svih stanja, možemo definirati skup svih akcija. Okolina ima definirane vjerojatnosti da će iz nekog stanja pod djelovanjem neke akcije prijeći u neko sljedeće stanje te da će agentu isporučiti odgovarajuću nagradu.

Problem s kojim smo sada suočeni jest sljedeći: agentu na raspolaganju stoji implementacija okoline, kako smo je modelirali sučeljem `DiscreteEnvironment`: to je crna kutija nad kojom agent može napraviti resetiranje (postavljanje u početno stanje), može okolinu pitati u kojem se stanju nalazi te može okolini dojaviti da on tada obavlja određenu akciju. Okolina će na to reagirati; na neki će način promijeniti svoje stanje i agentu će dojaviti nagradu koju dobiva za taj potez. Nešto povoljniji scenarij bio bi kada bismo okolinu mogli postaviti u bilo koje stanje - no to općenito nije slučaj.

Primjetite u čemu je razlika između ove situacije i situacije koju smo imali u prethodnom poglavlju: mi (agent) sada okolinu gleda kao crnu kutiju. Agentu na raspolaganju više ne stoji model okoline. Agent ne zna što može očekivati od okoline, i jedino što mu preostaje jest da pokuša napraviti neku akciju, pa vidi što će se dogoditi. Primijetimo odmah nezgodnu posljedicu: sada više ne možemo iz nekog stanja probati jednu akciju, pa dugu akciju, pa treću akciju, i tako redom, pa onda odabrati koju ćemo stvarno izvesti. Razlog tome jest činjenica da čim odaberemo prvu akciju, okolina će je izvesti i potencijalno promijeniti stanje; sada ne možemo napraviti *undo* (*"draga okolino, ja sam se samo šalio, ajde molim te vrati se u prethodno stanje"*) kako bismo isprobali sljedeću akciju.

Kako u takvom scenariju agent može naučiti optimalnu politiku - jer to je ono što nas zapravo zanima?

Prvi pristup koji ćemo opisati naziva se učenje temeljeno na modelu (engl. *model-based learning*). Evo ideje: odaberimo za agenta neku početnu politiku koju će agent slijediti (tu politiku možemo generirati slučajno). Uz tu politiku prethodno opisanim postupkom uzorkovanja možemo odrediti vrijednosti stanja: naprosto ćemo igru ponavljati puno puta. Štoviše, na temelju svake odigrane epizode možemo ažurirati vrijednosti svih stanja kroz koja smo prošli tijekom te

epizode, ako smo negdje pamtili nagrade primljene u svakom koraku. Jednom kada smo procijenili vrijednosti stanja, htjeli bismo popraviti politiku koju je agent slijedio - kao kod iteracije politike, htjeli bismo provesti ažuriranje politike. Prisjetimo se izraza (2.7) na temelju kojeg ažuriramo politiku:

$$\pi^*(s) = \arg\max_a \left(\sum_{s'} p(s'|s,a) \left[r(s,a,s') + \gamma \cdot v^*(s') \right] \right)$$

Da bismo za svako stanje mogli odrediti akciju koju želimo napraviti nužan nam je model okoline: trebamo vjerojatnosti $p(s'|s,a)$ te nagrade $r(s,a,s')$. S obzirom da ovo nemamo, agent bi prilikom igranja dinamički trebao graditi ovaj model. Trebao bi pamtiti sve odigrane epizode, za svako stanje koju je akciju odabrao u koliko od slučajeva, te koliko je puta okolina pod tom akcijom prešla u koje stanje i koju smo nagradu dobili. Na temelju tih podataka možemo procijeniti potrebne vjerojatnosti pa tu procjenu koristiti za ažuriranje politike. Jednom kad smo ažurirali politiku, možemo krenuti sve ciklički ponavljati - puno puta igramo, procjenjujemo vrijednosti stanja, pamtimo sve što se dogodilo pa ažuriramo model okoline koji gradimo.

Opisani postupak može biti računski vrlo zahtjevan, no omogućava agentu koji ne zna kako se okolina ponaša, da izgradi model te okoline na temelju mnoštva epizoda koje radi s okolinom, pa na temelju tog modela da popravlja svoju politiku koju koristi prilikom interacije s okolinom.

Drugi pristup naziva se izravno učenje ili učenje koje ne gradi model (engl. *model-free learning*) i taj pristup uobičajeno razmatramo kada govorimo o podržanom učenju.

3.1 Učenje vrijednosti

Ako agent ne raspolaže modelom okoline i ne želi ili mu nije praktično izgraditi model okoline, agent postoji način na koji može vrijednosti stanja naučiti relativno učinkovito izravno kroz interakciju s okolinom. Prisjetimo se: vrijednost stanja uz optimalnu politiku odgovara iznosu nagrade koji će agent dobiti ako u tom stanju odigra optimalnu akciju, plus korigiranu sumu nagradi koju će dobiti nastavkom optimalnog igranja iz sljedećeg stanja.

Za našeg agenta to znači da kad se zatekne u nekom stanju s, može odabrati akciju a, izvršiti tu akciju, od okoline primiti nagradu r i pogledati u koje je stanje okolina prešla: s'. S obzirom da agent čitavo vrijeme bilježi svoje procjene vrijednosti stanja, sada agent može pogledati:

1. kolika je njegova trenutna procjena vrijednosti stanja $v(s)$,
2. što je na temelju interakcije s okolinom otkrio da bi vrijednost stanja trebala biti: $g = r + \gamma \cdot v(s')$, pri čemu pretpostavlja da je njegova procjena vrijednosti stanja s' ispravna; ovu vrijednost zvat ćemo ciljnom vrijednosti.

I sada agent treba korigirati svoju procjenu za što se uobičajeno koristi linearna interpolacija.

R **Linearna interpolacija.** Neka su a i b dva broja. Linearna interpolacija $l(\alpha;a,b)$ je funkcija koja ovisno o parametru $\alpha \in [0,1]$ vraća vrijednost koja je između ta dva broja, pri čemu je $l(0;a,b) = a$, $l(1;a,b) = b$ a za $0 < \alpha < 1$ kako α ide od 0 prema 1, tako se vrijednost interpolacije mijenja od a do b, i to linearno. Linearnu interpolaciju možemo zapisati na dva uobičajena načina:

$$l(\alpha;a,b) = (1-\alpha) \cdot a + \alpha \cdot b$$

ili kao:

$$l(\alpha;a,b) = a + \alpha \cdot (b-a)$$

Primjerice, za $\alpha = 0.5$, linearna interpolacija vraća upravo aritmetičku sredinu između a i b.

Agent će svoju procjenu vrijednosti stanja $v(s)$ stoga korigirati prema izrazu koji interpolira između trenutne vrijednosti i ciljne vrijednosti:

$$v(s) \leftarrow (1-\alpha)\cdot v(s) + \alpha \cdot g$$

odnosno koristeći izraz:

$$v(s) \leftarrow (1-\alpha)\cdot v(s) + \alpha \cdot \left(r(s,a,s') + \gamma \cdot v(s')\right) \tag{3.1}$$

Da bi agent naučio vrijednosti stanja uz fiksnu politiku igranja, trebao bi odigrati puno epizoda, i tijekom svake epizode koristeći izraz (3.1) nakon svakog koraka korigirati procjenu vrijednosti stanja. Primijetite da se time tijekom jedne epizode korigiraju procjene za sva stanja kroz koja je agent prošao u toj epizodi. Postupak bi trebalo pokretati iz svih neterminalnih stanja, a parametar α ne bi smio biti prevelik (štoviše, s trajanjem učenja vrijednost bi polako trebalo smanjivati, kako bi se omogućilo da procjene iskonvergiraju).

Opisani postupak pripada u porodicu algoritama učenja s vremenskom razlikom (engl. *temporal difference learning*), jer podatak za korak t korigira na temelju informacija koje je dobio u kasnijem vremenskom trenutku (u ovom primjeru, u koraku $t+1$).

Implementacija je prikazana u nastavku.

Pseudokod 3.1 — Algoritam učenja vrijednosti.

```
public static <S,A> double[] valueLearning(DiscreteEnvironment<S, A>
    world, BasicDiscreteEnvironmentModel<S, A> envModel, Policy<S, A
   > agentPolicy, double alpha, int iterLimit) {
  int n = envModel.getHighestStateIndex()+1;
  double[] values = new double[n];

  List<S> nonTerminalStates = envModel.getAllStates().stream().
     filter(s->!envModel.isTerminal(s)).collect(Collectors.toList())
     ;
  int startStateIndex = 0;

  for(int iteration = 0; iteration < iterLimit; iteration++) {
    world.setCurrentState(nonTerminalStates.get(startStateIndex++));
    if(startStateIndex >= nonTerminalStates.size()) startStateIndex
       =0;

    while(!world.isFinished()) {
      S currentState = world.getCurrentState();
      A action = agentPolicy.pickAction(currentState);
      double reward = world.applyAction(action);

      int currentStateIndex = envModel.getStateIndex(currentState);
      int newStateIndex = envModel.getStateIndex(world.
         getCurrentState());

      double expectedValue = reward + envModel.getGamma()*values[
         newStateIndex];

      values[currentStateIndex] = (1-alpha)*values[currentStateIndex
         ] + alpha*expectedValue;
    }
  }
```

```
    return values;
  }
```

Ovaj postupak nažalost ne možemo iskoristiti da bismo omogućili ažuriranje politike koju agent koristi, kako bi agent postao bolji. Sjetite se: da bismo odredili bolju politiku, nisu nam dovoljne vrijednosti stanja (ako nemamo model okoline), već trebamo q-vrijednosti stanja (drugim riječima: trebamo u svakom stanju informaciju koliko će biti dobro ako u tom stanju odigramo određenu akciju).

Isprobajte

Algoritam učenja vrijednosti možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example01.MainValueLearning`
`rl.example02.MainValueLearning`
U oba slučaja radi se o agentu koji poteze bira potpuno slučajno; uvjerite se da dobivene vrijednosti stanja odgovaraju vrijednostima koje smo već prethodno utvrdili uzorkovanjem.

3.2 Algoritam SARSA

Algoritam SARSA je prvi od algoritama koji će omogućiti da agent kroz učenje istovremeno popravlja svoju politiku. U prethodnom poglavlju pokazali smo kako agent može uz fiksnu politiku naučiti vrijednosti stanja; koristili smo ažuriranje prema izrazu:

$$v(s) \leftarrow (1-\alpha)\cdot v(s) + \alpha \cdot \big(r(s,a,s') + \gamma \cdot v(s')\big)$$

što nam nije bilo dovoljno da bismo mogli ažurirati politiku. Stoga ćemo, umjesto da učimo vrijednosti stanja, učiti direktno q-vrijednosti. Svaku epizodu možemo zapisati nizom oblika:
$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, s_3, a_3, r_4, s_4, \ldots, s_{T-1}, a_{T-1}, r_T, s_T$
što možemo pročitati na sljedeći način: bili smo u stanju s_0, izvšili akciju a_0, dobili smo nagradu r_1 i prešli u stanje s_1, u tom smo stanju izvršili akciju a_1, dobili smo nagradu r_2 i prešli u stanje s_2, i tako dalje, sve do trenutka T u kojem smo kao odgovor na akciju a_{T-1} dobili posljednju nagradu r_T i prešli u terminalno stanje s_T.

S obzirom da želimo učiti q-vrijednosti stanja, prisjetimo se kako su iste definirane (uz pretpostavku da okolina za jednu akciju uvijek prelazi u isto stanje; u suprotnom izraz treba raspisati po svim mogućim stanjima uz odgovarajuće vjerojatnosti):

$$q(s,a) = r(s,a,s') + \gamma \cdot v_\pi(s')$$

Kako nemamo na raspolaganju procjene vrijednosti stanja, a učimo q-vrijednosti, algoritam SARSA član $v_\pi(s')$ aproksimira s $q(s',a')$; drugim riječima, algoritam slijedno "kližući" po zapisu epizode gleda petorke

$$(s_t, a_t, r_{t+1}, s_{t+1}, a_{t+1}) = (s,a,r,s',a')$$

te na temelju svake takve petorke ažurira procjenu q-vrijednosti prema izrazu:

$$q(s,a) \leftarrow (1-\alpha)\cdot q(s,a) + \alpha \cdot \big(r(s,a,s') + \gamma \cdot q(s',a')\big)$$

Time je jasno odakle naziv algoritma: *State-Action-Reward-State-Action*. Ažuriranje se radi za svako neterminalno stanje, a u terminalnom stanju definicijski se uzima da je $q(s_T,a) = 0$.

Prilikom igre, agent želi igrati optimalno, što znači da će svoju politiku igranja temeljiti upravo na trenutnim procjenama q-vrijednosti: u stanju s htjet će odabrati onu akciju za koju se uvjerio da je najbolja, odnosno ima najveću q-vrijednost.

Primijetimo da ovim pristupom agent politiku gradi na temelju procjena q-vrijednosti koje istovremeno uči, što znači da svakim ažuriranjem procjena q-vrijednosti agent zapravo istovremeno ažurira i svoju politiku.

Opisani pristup, kada bismo ga implementirali direktno, nikada ne bi radio dobro. Naime, pogledajmo samo početnu situaciju: sve procjene q-vrijednosti smo postavili nasumično (ili smo ih sve postavili na nulu). Nemamo još nikakvih saznanja što je dobro, a što ne. U takvoj situaciji, graditi politiku igranja na q-vrijednostima (i to još pohlepno: biramo akciju s najvećom q-vrijednosti) nema smisla.

Da bi dani algoritam omogućio agentu da se učenjem popravlja, trebamo malo modificirati politiku koju agent koristi. Konkretno, u ranim fazama učenja agent bi trebao više istraživati: ignorirati trenutne q-vrijednosti i akcije birati nasumično, da vidi kako će okolina reagirati. Kako učenje napreduje, agent bi trebao smanjivati istraživanje i više se fokusirati na iskorištavanje do tada naučenoga. Ovaj balans ranog istraživanja i kasnijeg iskorištavanja bitan je kako bi se osiguralo sve bolje i bolje ponašanje agenta.

Opisano ponašanje moguće je ostvariti na više načina; jedan od njih je uporabom ε-pohlepne politike. ε-pohlepna politika je politika pri kojoj s vjerojatnošću ε agent odabire potpuno nasumično koju će akciju izvršiti, a s vjerojatnošću $(1-\varepsilon)$ bira onu akciju koja u promatranom stanju ima maksimalnu q-vrijednost. Ako takvih ima više, ona potpuno slučajno bira koju će od tih maksimalnih izvesti.

Prilikom postupka učenja, u ranim fazama parametar ε trebao bi biti velik (1 ili blizak). Kako učenje napreduje, vrijednost parametra ε trebalo bi polagano smanjivati, kako bi agent češće birao akcije za koje je kroz istraživanje naučio da su dobre.

Implementacija ovog algoritma dostupna u priloženom projektu. Način odabira akcije prilikom učenja modeliran je sučeljem `ActionPicker`, a dostupna je implementacija koje uvijek posve slučajno odabiru akciju (razred `RandomActionPicker`), implementacija ε-pohlepne politike uz fiksan ε (razred `EpsilonGreedyActionPicker`) te implementacija ε-pohlepne politike uz ε koji se tijekom učenja linearno mijenja od zadanog početnog do zadanog konačnog (razred `EpsilonDecayingGreedyActionPicker`).

Opisani postupak učenja možemo još malo doraditi, što nas vodi na algoritam opisan u sljedećem odjeljku.

Isprobajte

Algoritam SARSA možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example01.MainSARSALearning`
`rl.example02.MainSARSALearning`
U oba slučaja radi se o agentu koji poteze bira potpuno slučajno; uvjerite se da dobivene vrijednosti stanja odgovaraju vrijednostima koje smo već prethodno utvrdili uzorkovanjem.

3.3 Q-učenje

Algoritam Q-učenje (engl. *Q-learning*) popularan je i robustan algoritam koji omogućava da agent nauči optimalnu politiku igranja, pri čemu tijekom učenja može čitavo vrijeme igrati koristeći neku drugu politiku (ali ne mora).

Za razliku od algoritma SARSA, algoritam Q-učenje vraća se na početni izraz:

$$q(s,a) = r(s,a,s') + \gamma \cdot v_\pi(s')$$

te koristi činjenicu da je:

$$v_\pi(s') = \max_{a'} q(s',a')$$

čime ažuriranje q-vrijednosti radi prema izrazu:

$$q(s,a) \leftarrow (1-\alpha) \cdot q(s,a) + \alpha \cdot \left(r(s,a,s') + \gamma \cdot \max_{a'} q(s',a') \right) \tag{3.2}$$

Kod ovog pristupa, optimalnu vrijednost stanja s' procjenjujemo na temelju svih procijenjenih q-vrijednosti akcija u stanju s'.

Algoritam ima odlično svojstvo da tako dugo dok je osigurano da se dovoljno istražuje te da je stopa učenja mala (ali ne premala), algoritam će otkriti optimalne q-vrijednosti neovisno o politici koju agent koristi. Prilikom implementacije algoritma uobičajeno ipak koristimo politiku koja se jednim dijelom temelji i na trenutno naučenim vrijednostima (poput ε-pohlepne politike).

Prigramska implementacija prikazana je u nastavku.

Pseudokod 3.2 — Algoritam Q-učenje.

```
public static <S,A> double[][] qLearning(DiscreteEnvironment<S, A>
    world, BasicDiscreteEnvironmentModel<S, A> envModel, S startState
    , double alpha, int iterLimit, ActionPicker<A> picker) {
  int sn = envModel.getHighestStateIndex()+1;
  int an = envModel.getHighestActionIndex()+1;

  @SuppressWarnings("unchecked")
  A[] allActions = (A[])new Object[an];
  envModel.getAllActions().toArray(allActions);

  double[][] qtable = new double[sn][an];

  for(int iteration = 0; iteration < iterLimit; iteration++) {
    picker.episodeStarted();
    world.setCurrentState(startState);
    while(!world.isFinished()) {
      S currentState = world.getCurrentState();
      int currentStateIndex = envModel.getStateIndex(currentState);

      A action = picker.pickAction(currentStateIndex, allActions,
          qtable);
      int actionIndex = envModel.getActionIndex(action);

      double reward = world.applyAction(action);

      int newStateIndex = envModel.getStateIndex(world.
          getCurrentState());
```

```
      double[] qValues = qtable[newStateIndex];
      double qmax = qValues[0];
      for(int i = 1; i < qValues.length; i++) {
        qmax = Math.max(qmax, qValues[i]);
      }
      double expectedQValue = reward + envModel.getGamma()*qmax;

      qtable[currentStateIndex][actionIndex] = (1-alpha)*qtable[
          currentStateIndex][actionIndex] + alpha*expectedQValue;
    }
  }

  return qtable;
}
```

Isprobajte

Algoritam Q-učenja možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite programe ostvarene razredima:
`rl.example01.MainQLearning`
`rl.example02.MainQLearning`
Primijetite da su rezultati tipično bolji od onih postignutih algoritmom SARSA.

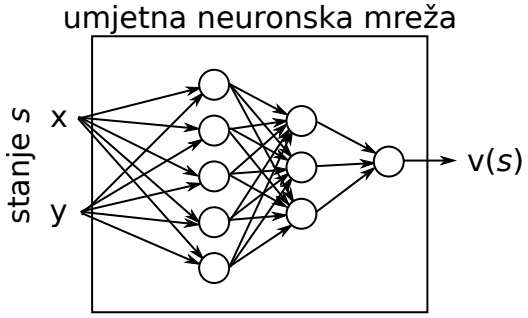
3.4 Pristupi za kontinuirane i/ili prevelike prostore stanja

U čitavom dosadašnjem razmatranju radili smo uz pretpostavku da je okolina diskretna. Implicitna pretpostavka (koju nikada nismo na glas izrekli) jest i da broj stanja nije prevelik, tako da sve relevantne informacije možemo pamtiti u tablicama u memoriji. Primjerice, u jednodimenzijskom polju decimalnih brojeva pamtili smo sve vrijednosti stanja (a polje smo indeksirali po rednom broju stanja); u dvodimenzijskom polju decimalnih brojeva pamtili smo sve q-vrijednosti.

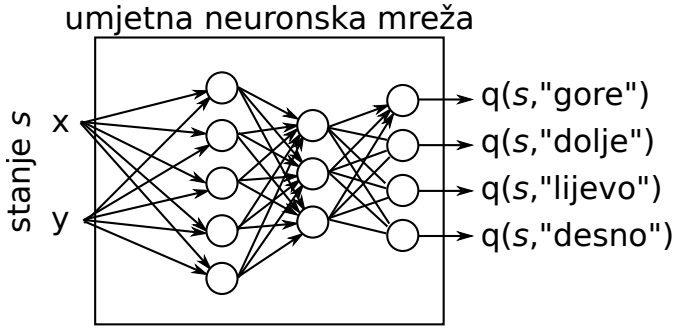
Ovaj pristup očito neće funkcionirati ako je stanje kontinuirano (primjerice, jedna od komponenta stanja je brzina kojom nam se neprijatelj približava; brzina je kontinuirana vrijednost). Drugi primjer kada ovo neće funkcionirati jest ako je broj stanja velik, primjerice 10^{50}. Konačno, nekada nam nije niti u interesu da razlikujemo sva stanja: zamislite jedan rešetkasti svijet i situaciju gdje je agent na polju (i,j) a neprijatelj na polju $(i+3,j+3)$ - akcija koju bi agent trebao poduzeti vjerojatno ne ovisi o konkretnim i i j; situaciju gdje je agent na (50,50) i neprijatelj na (53,53) želimo tretirati isto kao i situaciju gdje je agent na (20,30) a neprijatelj na (23,33). Iako su u našoj igri to doista *različita stanja*, htjeli bismo ih tretirati isto kako bismo agentu omogućili da nauči generalizirati.

Sposobnost generalizacije posebno je važna jer ako je prostor stanja prevelik, tada je nerealno i očekivati da će algoritam učenja puno puta uspjeti posjetiti sva stanja i naučiti kako u svakom od njih optimalno odigrati.

Problem prevelikih tablica možemo riješiti tako da umjesto pohrane svih podataka u memoriju razvijemo aproksimator koji na ulazu dobiva informaciju o stanju, a na izlazu generira traženu vrijednost. Kao aproksimator bismo pri tome mogli koristiti bilo kakav aproksimacijski model - primjerice, umjetnu neuronsku mrežu. Slika u nastavku prikazuje aproksimaciju vrijednosti stanja u rešetkastom svijetu. Kao ulaz u mrežu šaljemo informaciju o stanju: x-koordinatu i y-koordinatu ćelije čija nas vrijednost zanima; mreža računa i na izlazu generira vrijednost tog stanja.



Radimo li s q-vrijednostima, tada bismo koristili aproksimacijski model bi imao više izlaza: za svaku od akcija postojao bi jedan izlaz koji bi generirao q-vrijednost odgovarajuće akcije. Slika u nastavku ovo ilustrira, ponovno na primjeru rešetkastog svijeta gdje su moguće akcije "gore", "dolje", "lijevo" i "desno".



Primijetimo da smo ovime riješili dva problema: problem prevelikog prostora stanja, kao i problem kontinuiranih varijabli stanja; aproksimacijski modeli poput umjetnih neuronskih mreža mogu raditi s decimalnim brojevima na ulazima.

Prilikom učenja, kada algoritam treba promijeniti vrijednost stanja ili q-vrijednost, sada to više ne može napraviti tako da novu vrijednost "upiše" u tablicu jer tablica ne postoji. Umjesto toga, odredit ćemo koju vrijednost daje model, a koja je nova željena vrijednost. Razlika tih vrijednosti je pogreška, i potom ćemo korigirati model (u slučaju umjetne neuronske mreže korigirat ćemo težine između neurona) kako bi se ta pogreška smanjila (primjerice, na način kako to radi algoritam propagacije pogreške unatrag, koji je temeljni algoritam učenja unaprijednih umjetnih neuronskih mreža kakve smo ovdje pokazali).

Alternativni pristup koji se koristi jest uporaba vektora značajki.

3.4.1 Reprezentacija stanja značajkama

Ako je prostor stanja prevelik, ili ako postoji mnoštvo različitih stanja u kojima bi se agent trebao ponašati na jednak način, umjesto izravne uporabe stanja modele podržanog učenja možemo pisati tako da koriste značajke stanja (engl. *features*).

Značajka stanja $f_i(s)$ je funkcija koja prima stanje i vraća odgovaraću skalarnu vrijednost. Primjerice, za naš prvi rešetkasti svijet, ako s r označimo redak u kojem se nalazi agent, a sa s stupac u kojem se nalazi agent, mogli bismo definirati sljedeće značajke:

- $f_1(s) = r$
- $f_2(s) = c$
- $f_3(s) = r \cdot c$
- $f_4(s) = |3 - r - c|$
- $f_5(s) = |r - c|$

- $f_6(s) = |3-r-c| \cdot |r-c|$
- $f_7(s) = \sqrt{|3-r-c| \cdot |r-c|}$

Sada bismo vrijednost stanja mogli aproksimirati na temelju ovih značajki stanja (primjerice, umjetnoj neuronskoj mreži na ulaz bismo dostavili vrijednosti svih prethodno definiranih značajki određenog stanja, a mreža bi na izlazu dala vrijednost stanja).

Kako ovo ne bi ostalo previše apstraktno, idemo pogledati jedan puno jednostavniji aproksimacijski model: pretpostavimo da aproksimaciju obavlja afina transformacija (što je konceptualno linearna transformacija plus pomak); drugim riječima, pretpostavimo da je vrijednost stanja moguće izračunati prema izrazu:

$$v(s) = \beta_1 \cdot f_1(s) + \beta_2 \cdot f_2(s) + \beta_3 \cdot f_3(s) + \beta_4 \cdot f_4(s) + \beta_5 \cdot f_5(s) + \beta_6 \cdot f_6(s) + \beta_7 \cdot f_7(s) + \beta_8 \quad (3.3)$$

gdje su β_1 do β_8 8 koeficijenata (realni brojevi).

Prilikom učenja vrijednosti, u koraku gdje smo iz t prešli u $t+1$, iz stanja s otišli smo u stanje s' odabirom akcije a, pa smo ažuriranje radili prema izrazu (3.1) do kojeg smo došli iz izraza:

$$v(s) \leftarrow (1-\alpha) \cdot v(s) + \alpha \cdot g$$

gdje je $g = r(s,a,s') + \gamma \cdot v(s')$, odnosno nova očekivana vrijednost stanja. Kod podsjetnika na linearnu interpolaciju već smo pokazali da se prethodni izraz može zapisati i u obliku:

$$v(s) \leftarrow v(s) + \alpha \cdot (g - v(s))$$

odnosno:

$$v(s) \leftarrow v(s) + \alpha \cdot error(s)$$

gdje je $error(s) = g - v(s) = r(s,a,s') + \gamma \cdot v(s') - v(s)$, dakle pogreška, odnosno razlika između očekivane vrijednosti stanja izračunate u trenutku $t+1$ i trenutno pohranjene vrijednosti stanja. Ako vrijednosti stanja pamtimo u tablici, tada ovu korekciju možemo izravno ažurirati u tablici i nastaviti sa sljedećim korakom. Ako. međutim, vrijednosti stanja ne pamtimo u tablici već ih aproksimiramo nekim modelom (u našem primjeru: afinom transformacijom), tada na neki način moramo ažurirati parametre modela (koeficijente β) kako bismo smanjili magnitudu pogreške. Stoga ćemo definirati pomoćnu funkciju:

$$E(s) = \frac{1}{2} error^2(s)$$

i nju ćemo minimizirati. Kako se funkcija mijenja ovisno o promjeni neke varijable govori nam parcijalna derivacija funkcije s obzirom na tu varijablu. Pogledajmo stoga čemu je jednaka $\frac{\partial E(s)}{\partial \beta_1}$:

$$\begin{aligned} \frac{\partial E(s)}{\partial \beta_1} &= \frac{\partial}{\partial \beta_1} \left(\frac{1}{2} (g - v(s))^2 \right) \\ &= \frac{1}{2} \cdot (g - v(s)) \cdot (-1) \cdot \frac{\partial v(s)}{\partial \beta_1} \end{aligned}$$

Kako je:

$$\begin{aligned} \frac{\partial v(s)}{\partial \beta_1} &= \frac{\partial}{\partial \beta_1} (\beta_1 \cdot f_1(s) + \beta_2 \cdot f_2(s) + \beta_3 \cdot f_3(s) + \beta_4 \cdot f_4(s) + \beta_5 \cdot f_5(s) + \beta_6 \cdot f_6(s) + \beta_7 \cdot f_7(s) + \beta_8) \\ &= f_1(s) \end{aligned}$$

slijedi da je:

$$\frac{\partial E(s)}{\partial \beta_1} = -(g - v(s)) \cdot f_1(s)$$

Na isti način možemo se uvjeriti da je za parametre β_i, $1 <= i <= 7$, dobivamo:

$$\frac{\partial E(s)}{\partial \beta_i} = -(g - v(s)) \cdot f_i(s)$$

dok za parametar β_8 istim postupkom dobivamo:

$$\frac{\partial E(s)}{\partial \beta_8} = -(g - v(s)).$$

Promijenimo li vrijednost parametra proporcionalno parcijalnoj derivaciji uz pozitivan koeficijent proporcionalnosti, vrijednost funkcije će se povećati. Stoga ćemo svakom parametru promijeniti vrijednost tako da ga pomaknemo u suprotnom smjeru, uz koeficijent proporcionalnosti α, gjde je α mali pozitivan broj (npr. 0.01). Ažuriranje ćemo dakle raditi prema izrazu:

$$\beta_i \leftarrow \beta_i - \alpha \cdot \frac{\partial E(s)}{\partial \beta_i} \tag{3.4}$$

što u našem slučaju za koeficijente β_1 do β_7 daje:

$$\beta_i \leftarrow \beta_i + \alpha \cdot (g - v(s)) \cdot f_i(s) = \beta_i + \alpha \cdot (r(s,a,s') + \gamma \cdot v(s') - v(s)) \cdot f_i(s)$$

a za koeficijent β_8 daje:

$$\beta_8 \leftarrow \beta_8 + \alpha \cdot (g - v(s)) = \beta_8 + \alpha \cdot (r(s,a,s') + \gamma \cdot v(s') - v(s))$$

Ponovimo još jednom: ako umjesto tabličnog pohranjivanja vrijednosti stanja koristimo aproksimacijski model, tada ćemo:

1. u svakom koraku pitati model da nam odredi $v(s)$ (ako koristimo značajke, na temelju stanja odredit ćemo sve značajke i njih predati modelu; ako govorimo o modelu iz danog primjera, model će ih izmnožiti koeficijentima, zbrojiti i vratiti aproksimiranu vrijednost)
2. u svakom koraku pitati model da nam odredi $v(s')$ (ako koristimo značajke, na temelju stanja odredit ćemo sve značajke i njih predati modelu; ako govorimo o modelu iz danog primjera, model će ih izmnožiti koeficijentima, zbrojiti i vratiti aproksimiranu vrijednost)
3. na temelju primljene nagrade r i aproksimiranih $v(s)$ i $v(s')$ možemo korigirati parametre modela
4. ovaj postupak ponavljat ćemo za svako stanje u epizodi te iterativno za mnoštvo epizoda.

Implementacija opisanog postupka dostupna je u priloženom projektu pri čemu su apstrahirani postupci vađenja značajki i aproksimacijski modeli (sve je modelirano kroz prikladna sučelja) tako da iste možemo mijenjati po potrebi (vidi metodu `featureBasedValueLearning`).

Razred `rl.example01.MainFeatureBasedValueLearning` predstavlja implementaciju ovog pristupa na prvom rešetkastom svijetu. U tom razredu dana je i implementacija objekta koji iz stanja vadi upravo opisanih 7 značajki. Primjer koristi afinu transformaciju značajki kao aproksimacijski model i uči vrijednosti stanja za agenta koji koristi politiku nasumičnog odabira akcije u svakom stanju. Po pokretanju, program će ispisati značajke za sva stanja, potom odraditi postupak učenja, ispisati za sva stanja naučene vrijednosti stanja te ispisati sam aproksimacijski model, kako biste vidjeti koje vrijednosti imaju pojedini koeficijenti.

Osvrnimo se na prednost uporabe aproksimacijskih modela: umjesto pamćenja svih zapisa tablice (u slučaju velikih prostora stanja, ovo može biti i nemoguće), svo naučeno znanje sažeto

je u nekoliko realnih brojeva koji predstavljaju koeficijente korištenog aproksimacijskog modela. Dodatno, kako aproksimacijski modeli zapravo rade sažimanje informacija (prisiljeni su generalizirati), za očekivati je da će model za slična ali različita stanja dati sličnu vrijednost na izlazu, što je često poželjno svojstvo. Problem koji se može javiti jest nedovoljno ekspresivan model: ako je funkcija koju aproksimiramo presložena za model koji smo odabrali. U tom slučaju postupak učenja će završiti s visokom pogreškom.

Opisani pristup možemo koristiti i za aproksimaciju q-vrijednosti, pri čemu u tom slučaju značajke mogu biti općenito funkcije stanja i akcije koja se razmatra.

Aproksimacijske modele te reprezentaciju stanja temeljenu na značajkama možemo iskoristiti i za učenje optimalne politike agenta. Čitatelju se ostavlja za vježbu da napravi modifikaciju algoritma Q-učenja tako da za pohranu q-vrijednosti ne koristi tablicu već aproksimacijski model.

Isprobajte

Uporabu značajki i aproksimacijskog modela možete isprobati i samostalno. U pripremljenom projektu pronađite pa pokrenite program ostvaren razredom:
`rl.example01.MainFeatureBasedValueLearning`
Radi se implementaciji algoritma učenja vrijednosti koji međutim pohranu ne radi u tablicu već koristi aproksimacijski model koji obavlja afinu transformaciju nad značajkama stanja.

3.5 Daljnji pristupi

Kroz prethodni tekst dali smo samo kratak uvod u algoritme podržanog učenja. Zainteresiranom čitatelju skrećemo pažnju da danas postoji još niz drugih algoritama, čak i u kombinaciji s modelima dubokog učenja kojima se postižu izvrsni rezultati. Opis istih izlazi iz okvira tema koje su odabrane za kolegij Umjetna inteligencija, pa ih ovdje dalje nećemo razrađivati.

Bibliografija

Knjige

Članci

Konferencijski radovi i ostalo