UNIVERZA V LJUBLJANI
FAKULTETA ZA MATEMATIKO IN FIZIKO

Finančna matematika – 1. stopnja

Tim Kalan

Spodbujevalno učenje pri igranju namiznih iger

Delo diplomskega seminarja

Mentorica: prof. dr. Marjetka Knez

Ljubljana, 2021

Kazalo

Spodbujevalno učenje pri igranju namiznih iger

Povzetek

Motivacija za nalogo je bila razumeti algoritme, ki se učijo prek poskušanja in napak. Na začetku postavimo teoretični okvir v obliki Markovskih procesov odločanja. V nadaljevanju se posvetimo izpeljavi in opisu metod, ki temeljijo na konceptu dinamičnega programiranja. Te metode potem posplošimo in predstavimo tri glavne iterativne algoritme: Monte Carlo, TD(0) in TD(λ). Ker pa smo želeli ustvariti kompetentnega igralca namiznih iger, te pa imajo pogosto veliko količino stanj, se posvetimo še funkcijski aproksimaciji in kombinaciji nevronskih mrež s predstavljenimi algoritmi.

V drugem delu naloge si bolj natančno ogledamo kombinatorne igre; to je teoretični model za namizne igre. Nato opišemo nekaj pomembnih razlik, do katerih pride pri spodbujevalnem učenju v tem konteksu in si ogledamo, kako se prilagodi koncept optimalne strategije in vrednostne funkcije.

V zadnjem delu apliciramo teorijo še na praktičnem primeru. Na m,n,k-igrah uporabimo opisane algoritme in komentiramo njihovo učinkovitost.

Reinforcement learning in board games

Abstract

The motivation for this work is trying to understand algorithms that learn through trial and error. At the beginning we set the theoretical foundation by examining Markov decision processes. We then derive and describe methods, which are based on dynamic programming. Further we generalize these methods and present three iterative algorithms: Monte Carlo, TD(0) and TD(λ). Since we want to create a competent board game player, and board games often have a large number of states, we observe also the function approximation and combine neural networks with the described algorithms.

In the second part we examine combinatorial games in more detail. This is our theoretical model for board games. We then describe some important differences which have to be made to reinforcement learning in this context and look at how to adjust the concept of optimal strategies and value functions.

In the last part we apply the presented theory to a practical example. We use the described algorithms to solve some m,n,k-games and comment on their efficiency.

Math. Subj. Class. (2010): 62L20, 68T05, 90C40

Ključne besede: spodbujevalno učenje, Markovski proces odločanja, učenje s časovno razliko, po-stanja, samoigra

Keywords: reinforcement learning, Markov decision process, temporal-difference learning, afterstates, self-play

1. Uvod

Namizne igre ljudje igramo že od prazgodovine. Na Kitajskem je bila igra go znana kot ena izmed štirih umetnosti kitajskega učenjaka poleg igranja inštrumenta s strunami, kaligrafije in slikanja. Spremljajo nas že zelo dolgo časa, zato je naravno, da jih želimo ljudje čim bolje igrati.

Z razvojem računalnika in računalništva je bil ta problem postavljen v novi luči. Vprašanje ni bilo več samo, kako dobro lahko človek igra igro sam, temveč tudi do kakšnega nivoja lahko spravi računalnik. Izkazalo se je, da nam pri tem problemu (in mnogih drugih) zelo dobro koristi »umetna inteligenca« oz. metode strojnega učenja. Eno izmed vej strojnega učenja bomo predstavili v tem delu in pogledali, kako nam lahko pomaga pri igranju namiznih iger.

Ideja, da bi nek stroj igral igre, ni nova, in kompleksnosti takega stroja so se zavedali ljudje že pred obstojem računalnika. Za konec uvodnega dela morda zabeležimo še citat iz eseja ameriškega pisatelja in pesnika Edgarja Allana Poea, ki govori o mehaničnem igralcu šaha:

> »Če prej omenjenemu [igralcu šaha] rečemo čisti stroj, moramo biti pripravljeni priznati, da je zunaj vseh primerjav, najbolj čudovit izum človeštva.«

1.1. **Motivacija.** Motivacija za spodbujevalno učenje izhaja iz psihologije. Znana psihologa Thorndike in Skinner sta na živalih izvajala eksperimente. Postavila sta jih v neko novo situacijo, kjer je lahko žival naredila akcijo, ki je rezultirala v neki nagradi. Ko je bila žival ponovno postavljena v to situacijo, je hitreje ugotovila, katero akcijo mora storiti, da pride do nagrade.

Koncept, ki je opisan v zgornjem odstavku, se imenuje instrumentalno pogojevanje. Z njim se srečamo tudi ljudje; tako se namreč učijo otroci, odrasli ljudje pa se bolj zanesejo na logično razmišljanje. Vseeno pa je to motiviralo utemeljitelje spodbujevalnega učenja.

1.2. **Strojno učenje.** To relativno novo raziskovalno področje se deli na tri glavne veje:

- **Nadzorovano učenje** se ukvarja s tem, kako iz nekih označenih podatkov naučimo računalnik, da prepoznava razne signale (slike, govor, tekst, ...) in to znanje uporabi za razpoznavo novih, neoznačenih podatkov.
- **Nenadzorovano učenje** odstrani označevanje iz podatkov in v njih poskuša odkriti skrite vzorce.
- **Spodbujevalno učenje** se ukvarja z »učenjem iz izkušenj«.

1.3. **Struktura naloge.** Naloga je razdeljena na štiri glavne dele. Na začetku so predstavljeni osnovni koncepti spodbujevalnega učenja in nekateri glavni algoritmi s tega področja. Potem se osredotočimo na namizne igre in ob nekaj malega teorije iger povzamemo osnovne koncepte, na katere naletimo. V naslednjem odseku združimo znanje iz prejšnjih dveh in predstavimo, kako nam teorija iger pripomore pri spodbujevalnem učenju v tem konteksu. Na koncu pa so predstavljeni nekateri empirični rezultati, ki sledijo iz zgoraj navedene teorije.

2. Spodbujevalno učenje

Spodbujevalno učenje se ukvarja s t. i. učenjem iz interakcije oz. izkušenj. Čeprav se to na prvi pogled ne zdi kot računska metoda, pač pa stvar psihologije, bomo kmalu dognali, kako prevesti to idejo v računalniku razumljiv jezik.

2.1. Osnovni koncepti.

V osnovi nas zanima precej preprosta stvar: kako preslikati neko opazovano situacijo v akcijo na tak način, da učenec maksimizira numerično nagrado. V kontekstu spodbujevalnega učenja učenca imenujemo **agent**. Pri učenju ne obstaja opazovalec, ki bi agentu povedal ali pa namignil, katere akcije so dobre, to mora ugotoviti sam, s poskušanjem in napakami. V tem dejstvu se skriva bistvena razlika med spodbujevalnim učenjem in ostalimi vejami strojnega učenja.

Še ena razlika tiči v pomembnosti časa pri spodbujevalnem učenju. Pri drugih oblikah strojnega učenja se ponavadi ukvarjamo s tabelaričnimi podatki, tu pa modeliramo dinamičen proces, zato je naravno, da je pomemben čas. Čeprav se ga da modelirati zvezno, je za naše namene dovolj, da ga opazujemo kot diskretne točke $t \in \{1, \ldots, T\}$, kjer T označuje končni čas (v splošnem je lahko seveda $T = \infty$).

2.1.1. *Nagrada.*

Prvi pomemben koncept pri spodbujevalnem učenju je nagrada. Kot smo že omenili, to za nas pomeni numerično vrednost, kjer pozitivno število indicira »pozitivno nagrado«, negativno pa »kazen«. S pomočjo tega koncepta formaliziramo *cilj* učenja. Edini cilj agenta je maksimizacija te nagrade, pri čemer je vredno omeniti, da na nagrado agent lahko vpliva samo s svojimi akcijami (ne more recimo spremeniti načina, po katerem dobi nagrado).

Posebej pomembno je na tem mestu poudariti, da akcije nimajo nujno neposredne nagrade. Te lahko pridejo v poljubnem kasnejšem časovnem obdobju. To je smiselno, če gledamo z vidika namiznih iger: pri šahu ne razmišljamo samo o neposrednih akcijah, temveč razvijamo neko dolgoročno strategijo, ki nas na koncu nagradi z zmago.

Zgled 2.1 (Križci in krožci)**.** *Pri tej znani otroški igri (in pri mnogo drugih namiznih igrah) premik v času pomeni igranje poteze enega od igralcev, zato je diskreten čas popolnoma zadosten. Nagrado lahko modeliramo na preprost način: če zmagamo, prejmemo nagrado* 1*, če izgubimo pa* -1*. V vseh ostalih situacijah, torej za izenačenje in po vsaki potezi, prejmemo nagrado* 0*.*

Zavedati se moramo tudi potencialnih omejitev oz. pomanjkljivosti takega modela nagrad in ciljev.

Hipoteza 2.2 (Hipoteza o nagradi)**.** *Vse cilje je mogoče opisati kot maksimizacijo neke kumulativne numerične nagrade.*

Zgled 2.3 (Protiprimera hipotezi o nagradi)**.** *Problem je, da hipoteza dovoljuje samo enodimezionalnost:*

- *Ko kupujemo hamburger, nam je pomemben okus in cena; kaj nam več pomeni?*
- *Država želi med epidemijo ohraniti življenja in gospodarstvo; v kolikšni meri naj prioritizira ti dve kategoriji?*

Na tem mestu poudarimo še, da se da tudi v takih situacijah modelirati nagrado na zgoraj opisani način in da je ta koncept vseeno dovolj splošen, da zajame zelo velik razred problemov.

2.1.2. *Okolje.* Okolje predstavlja del našega sistema, na katerega agent nima nobenega vpliva. Funkcija okolja je, da agentu pokaže **stanje** (angl. *state*) in mu da nagrado glede na **akcijo**, ki jo prejme od njega. Če se ponovno osredotočimo na namizne igre, bi lahko rekli, da je okolje igralna plošča *in* nasprotnik – tudi nanj namreč nimamo vpliva. Okolje nam služi tudi kot sodnik akcij oz. stanj. V kontekstu programa za igranje iger torej okolje izbira nasprotnikove akcije, odloča katero stanje pomeni zmago in dodeljuje nagrade.

Zgled 2.4 (Križci in krožci)**.** *Okolje za nas pomeni* 3×3 *igralno polje* ***in*** *našega nasprotnika - to je torej človek ali nek algoritem (ki je lahko tudi kopija agenta).*

2.1.3. *Agent.* Kot smo že omenili, je agent naš učenec. Njegov cilj je torej maksimizacija numerične nagrade, to težnjo pa dosega s pomočjo **strategije** (angl. *policy*), ki mu pove, katero akcijo naj izbere v določenem stanju. Za ocenjevanje stanja si pomaga z **vrednostno funkcijo** (angl. *value function*). Kot ime implicira, je to funkcija, ki določa vrednosti stanjem (in akcijam).

Nagrada nam pove takojšnjo vrednost stanja, vrednostna funkcija pa to vrednost gleda na dolgi rok. Je izpeljanka nagrade, a veliko bolj primerna za maksimizacijo kot nagrada sama, saj upošteva, da so tudi stanja, ki ne prinesejo takojšnje nagrade lahko veliko vredna (na tem mestu se ponovno spomnimo šaha in grajenja strategije, ki nagrado prinese šele ob koncu igre).

Poleg tega je v splošnem lažje učenje preko vrednostnih funkcij kot preko strategij neposredno, saj je ponavadi stanj mnogokrat manj kot možnih strategij agenta.

Zapišimo zgoraj opisane pojme bolj formalno:

Definicija 2.5. Naj $\mathcal{S}$ označuje množico vseh stanj, $\mathcal{A}$ pa množico vseh akcij. Naj R_t, S_t, A_t zaporedoma označujejo slučajno nagrado, stanje in akcijo ob času t. Definiramo naslednje pojme:

- Agentova **strategija** (angl. *policy*) je preslikava, ki agentu pove, katero akcijo naj izbere v katerem stanju. Strategije delimo v dve skupini:
  - **Deterministična strategija** je preslikava $\pi : \mathcal{S} \to \mathcal{A}$, ki za vsako stanje s pove, katero akcijo a agent v njem izbere oz.
  $$\pi(s) = a.$$
  - **Stohastična strategija** je preslikava $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$, ki za vsako stanje s pove verjetnost, da se igra določena akcija a. To označimo z
  $$\pi(a|s) = P(A_t = a \mid S_t = s),$$
  kjer je t čas, v katerem je dosegljivo stanje s.

  Seveda lahko vsako deterministično strategijo predstavimo kot stohastično, kjer je verjetnost ene akcije 1, verjetnosti ostalih akcij pa so enake 0.
- Naj bodo $R_{t+1}, ..., R_T$ nagrade, ji jih bomo prejeli od trenutka t do končnega časa T. **Povračilo** (angl. *return*) G_t ob času t v splošnem definiramo za $T = \infty$,
  $$G_t = R_{t+1} + \gamma R_{t+2} + ... = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$
  kjer je $\gamma \in [0, 1]$ **diskontni faktor**. Predstavlja dejstvo, da imamo raje nagrade, ki bodo prišle prej. Formalno gledano je cilj učenja maksimizacija pričakovanega povračila.

- Naj bo π dana strategija agenta. **Vrednostna funkcija stanja** (angl. *state-value function*) glede na strategijo π je pogojno matematično upanje povračila, če začnemo v stanju s in se potem vedemo v skladu s strategijo π,

$$v_\pi(s) = \mathrm{E}[G_t \mid S_t = s]. \tag{1}$$

  Predstavlja torej pričakovani izplen, če se vedemo skladno s strategijo π.
- Naj bo π še vedno dana strategija agenta. **Vrednostna funkcija akcije** (tudi stanja-akcije) (angl. *action-value function*) glede na strategijo π je definirana kot

$$q_\pi(s, a) = \mathrm{E}[G_t \mid S_t = s, A_t = a]. \tag{2}$$

  Pove nam pričakovani izplen, če ob času t naredimo akcijo a, nato pa se vedemo skladno s strategijo π.

Tako v (1) kot tudi v (2) nam t označuje čas, v katerem je dosegljivo stanje s.

Zgled 2.6 (Križci in krožci)**.** *Agent je v tem primeru računalniški program, ki prejme igralno ploščo, nasprotnikove poteze in nagrade, nato pa prek poskušanja in učenja vrne optimalno strategijo, tj. za vsako stanje najboljšo možno akcijo.*

Deterministična strategija bi pomenila preslikavo, ki prejme reprezentacijo igralne plošče in vrne točno določeno potezo glede na to ploščo. Stohastična strategija pa pomeni preslikavo, ki sprejme isto stvar, a vrne neko verjetnostno porazdelitev med vsemi legalnimi potezami, ki jih ima agent (legalnost poteze je določena s strani okolja oz. pravil igre).

2.1.4. *Model.* Model je nenujen del našega sistema. Predstavlja znanje, ki ga ima agent o svojem okolju. Če imamo model, ga lahko uporabimo, da napovemo, kako se bo vedlo okolje in s tem premaknemo agentovo učenje iz čistih poskusov in napak na *načrtovanje* (angl. *planning*). Model je torej poleg strategije in vrednostne funkcije še tretja komponenta agenta. Na podlagi modela lahko agent »preračuna« smiselnost svojih akcij, brez da bi dejansko karkoli storil.

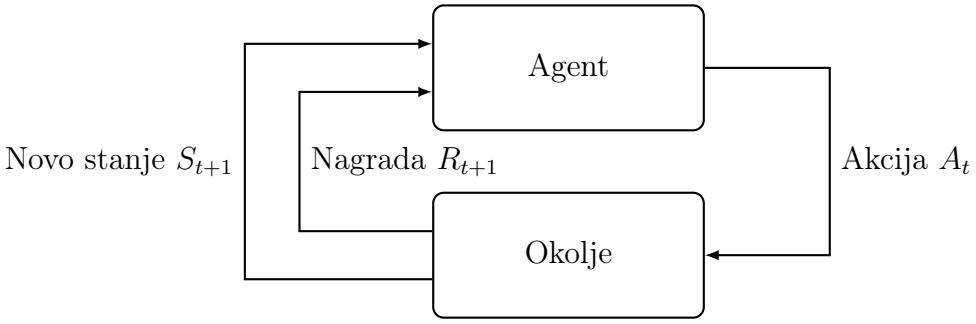
Prisotnost modela je glavna ločnica med dvema velikima, a zelo različnima vejama spodbujevalnega učenja. Če modela ni, govorimo o spodbujevalnem učenju brez modela (angl. *model-free reinforcement learning*), v nasprotnem primeru pa govorimo o učenju z modelom (angl. *model-based reinfocement learning*). Narava namiznih iger za dva igralca, kakršne obravnavamo tu, je, da lahko predvidimo, v kakšno stanje nas prinese naša akcija, zato imamo delni model okolja. Še vedno namreč ne poznamo strategije nasprotnika.

2.2. **Korak spodbujevalnega učenja.** Spodbujevalno učenje se pogosto ukvarja s procesi, ki naravno razpadejo v t. i. **epizode**. Tak proces so recimo namizne igre, kjer so epizode posamezne igre. Če bi poskušali robota naučiti hoje, bi lahko za epizode vzeli neko časovno okno ali pa morda hojo do prvega padca. Ni pa nujno, da je delitev tako naravna (ali pa sploh možna oz. smiselna). Za namene te diplomske naloge lahko privzamemo, da delitev na epizode obstaja.

Ideja učenja je, da agenta spustimo v okolje in mu dovolimo, da doživi (igra) mnogo epizod. Nato na nek način (s pomočjo spodbujevalnega učenja) ob nekih določenih časih (npr. po koncu posamezne epizode) posodobi svojo strategijo (in/ali vrednostno funkcijo).

Dejanski korak (npr. poteza enega od igralcev v namizni igri) v epizodi pa formalno gledano opredelimo na naslednji način.

- Agent naredi akcijo A_t ob prejetem stanju S_t in nagradi R_t.
- Okolje prejme akcijo A_t, posreduje agentu stanje S_{t+1} in nagrado R_{t+1}.



Slika 1. Zanka spodbujevalnega učenja

2.2.1. *Raziskovanje in izkoriščanje.* Eden izmed glavnih problemov, s katerim se srečamo pri spodbujevalnem učenju, je problem raziskovanja in izkoriščanja. Ko se agent uči, začne dojemati, katere akcije ali kombinacije akcij mu pripeljejo nagrado. Ko to ugotovi, seveda lahko začne te akcije *izkoriščati* in prejemati vso nagrado, ki akcijam pripada. Pri tem pa naletimo na problem. Če bi agent le izkoriščal te akcije, ne bi nikoli ugotovil, ali lahko katera druga akcija prinese še višjo nagrado. Tega ne bi izvedel, ker ne bi *raziskoval*. Če bi pa samo raziskoval, pa nikoli ne bi izkoristil potencialnih nagrad, ki jih sreča, torej se ne bi ničesar naučil.

Uravnoteženje raziskovanja in izkoriščanja je pomemben problem, a se izkaže, da ima dokaj enostavno rešitev (ki deluje dovolj dobro). Spoznali jo bomo v nadaljevanju.

Zgled 2.7 (Raziskovanje v namiznih igrah)**.** *Pri namiznih igrah agent lahko odkrije strategijo, ki premaga določenega nasprotnika. To strategijo lahko potem izrablja in nasprotnika vedno premaga, ko pa naleti na drugega nasprotnika se lahko izkaže, da je bila strategija učinkovita samo proti prvemu – naša strategija ni bila optimalna. Zato je pomembno, da tudi ob odkritju dobre strategije agent še vedno raziskuje prostor strategij. To najenostavneje dosežemo tako, da agenta prisilimo, da občasno igra naključne poteze.*

Morda se nekaterim bralcem zdi, da smo zaenkrat preveč »mahali z rokami«. To je zato, ker želimo, da se do te točke razvije intuicija o predstavljenih pojmih. V nadaljevanju bomo do sedaj opisane stvari bolj formalizirali.

2.3. **Markovski proces odločanja.** Spomnimo se najprej procesa spodbujevalnega učenja in ga poskusimo opisati bolj formalno: imamo zaporedje časovnih korakov $t = 0, 1, 2, \ldots$, ob katerih med sabo interaktirata agent in okolje. Ob koraku t agent prejme od okolja stanje (oz. reprezentacijo stanja) $S_t \in \mathcal{S}$, kjer $\mathcal{S}$ označuje množico vseh stanj. Na podlagi stanja in strategije, ki jo ima, izbere akcijo $A_t \in \mathcal{A}(S_t)$, kjer $\mathcal{A}(S_t)$ predstavlja množico akcij, ki jih ima agent na voljo v stanju S_t. Rezultat te akcije je nagrada $R_{t+1} \in \mathcal{R}$, kjer $\mathcal{R}$ označuje množico vseh nagrad, in novo stanje S_{t+1}.

Čeprav se da vse opisane koncepte posplošiti na števne in celo neštevne množice stanj in akcij, se bomo mi omejili na končne množice. To je pri namiznih igrah dovolj.

2.3.1. *Markovska veriga.* Dogajanje pri spodbujevalnem učenju lahko v grobem opišemo z zaporedjem slučajnih spremenljivk $S_0, S_1, \ldots$ v diskretnem času, to je, s slučajnim procesom stanj $(S_t)_{t=0}^T$. Zato je pomembno, da si natančno pogledamo nekaj lastnosti, ki jih lahko pričakujemo.

Definicija 2.8 (Markovska veriga)**.** Slučajni proces $(S_t)_{t=0}^T$ na končnem verjetnostnem prostoru (opremljenim z neko σ-algebro $\mathcal{F}$) $(\Omega, \mathcal{F}, P)$ je **Markovska veriga** oz. **Markovski proces** (angl. *Markov chain*), če zanj velja Markovska lastnost. To je lastnost, ki govori o pogojnih verjetnostih doseganja nekega stanja s_{t+1} ob času $t+1$ glede na celotno zgodovino procesa $S_0, \ldots, S_t$. Natančneje, Markovska lastnost zahteva, da je ta pogojna verjetnost enaka pogojni verjetnosti samo glede na trenutno stanje $S_t = s_t$ oz.

$$P(S_{t+1} = s_{t+1} \mid S_t = s_t, \ldots, S_0 = s_0) = P(S_{t+1} = s_{t+1} \mid S_t = s_t).$$

To verjetnost imenujemo *prehodna verjetnost* in jo označimo $p_{ss'} := P(S_{t+1} = s' \mid S_t = s)$. Če se ukvarjamo s končnim procesom, torej procesom, ki ima končno mnogo različnih stanj, lahko število stanj označimo z n. Potem lahko brez škode za splošnost vsa stanja označimo kar z $1, \ldots, n$. To nam potem dovoljuje, da prehodne verjetnosti zložimo v matriko $\mathcal{P} := [p_{ss'}]_{s,s' \in \mathcal{S}}$, ki ji pravimo *prehodna matrika.*

Zdaj Markovsko verigo predstavimo še na alternativni način: kot dvojico $(\mathcal{S}, \mathcal{P})$, kjer je $\mathcal{P}$ zgoraj definirana prehodna matrika, $\mathcal{S}$ pa množica vseh stanj.

Markovska lastnost pomeni, da je prihodnost neodvisna od preteklosti, če poznamo sedanjost. Spodbujevalno učenje se ukvarja predvsem s problemi, kjer to dejstvo drži. Tudi pri našem ciljnem problemu to načeloma velja: če pogledamo igralno ploščo na katerikoli točki, pogosto izvemo enako o trenutnem stanju, kot če bi opazovali igro od začetka.

2.3.2. *Markovski proces nagrajevanja.* Podoben koncept, ki se malo bolj približa dejanski situaciji v spodbujevalnem učenju, je *Markovski proces nagrajevanja.* Kot že ime morda namigne, je precej podoben Markovski verigi, le da v njem nastopajo *nagrade.* V tem modelu so tako kot stanja tudi nagrade slučajni proces $(R_t)_{t=1}^T$ na verjetnostnem prostoru $(\Omega, \mathcal{F}, P)$ z Markovsko lastnostjo.

Definicija 2.9 (Markovski proces nagrajevanja)**.** Pričakovani nagradi glede na stanje s pravimo **nagradna funkcija** (angl. *reward function*) in jo označimo

$$\mathcal{R}_s = E[R_{t+1} \mid S_t = s].$$

Naboru $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$, za katerega velja

- $\mathcal{S}$ je (končna) množica stanj,
- $\mathcal{P}$ je prehodna matrika $\mathcal{P}_{ss'} = P(S_{t+1} = s' \mid S_t = s)$,
- $\mathcal{R}$ je *nagradna funkcija* $\mathcal{R}_s = \mathrm{E}[R_{t+1} \mid S_t = s]$,
- $\gamma \in [0, 1]$ je *diskontni faktor*,

pravimo **Markovski proces nagrajevanja** (angl. *Markov reward process*).

Od navadne Markovske verige se torej razlikuje samo v prisotnosti nagrad ob vsakem koraku. Če te nagrade postavimo na $R_t = 0$ za vsak t, dobimo navadno Markovsko verigo. Pomembna razlika je še prisotnost diskontnega faktorja γ. Če velja $\gamma < 1$, potem procesu pravimo *diskontirani Markovski proces nagrajevanja.*

Diskontiranje je motivirano iz različnih vidikov: po eni strani nam pomaga, da se v primeru cikličnih procesov izognemo neomejenim povračilom. Poleg tega pa je

diskontiranje v mnogo pogledih naraven način za opis situacije: pogosto imamo raje nagrade, ki pridejo prej. Primer tega poznamo recimo iz ekonomije; denar, ki ga dobimo kasneje, nam pomeni manj, kot tisti, ki ga dobimo takoj.

Še vedno pa tovrsten proces ne opiše situacije, v kateri se znajdemo pri spodbujevalnem učenju, saj ne vsebuje koncepta akcij. Je pa že dovolj »globok«, da v njem lahko definiramo vrednostno funkcijo $v(s) = \mathrm{E}[G_t \mid S_t = s]$, ki je v tem primeru neodvisna od strategije π, saj strategija v tem modelu nima pomena.

Za vrednostno funkcijo lahko izpeljemo rekurzivno enačbo, s pomočjo katere lahko »rešimo« Markovski proces nagrajevanja. S tem mislimo, da vsakemu stanju dodelimo pravo vrednost. To je vrednost, ki jo stanju določata nagrada in nagradna funkcija:

$$\begin{aligned} v(s) &= \mathrm{E}[G_t \mid S_t = s] \\ &= \mathrm{E}[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s] \\ &= \mathrm{E}[R_{t+1} + \sum_{k=1}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s] \\ &= \mathrm{E}[R_{t+1} + \gamma(\sum_{k=1}^{\infty} \gamma^{k-1} R_{t+k+1}) \mid S_t = s] \\ &= \mathrm{E}[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\ &= \mathrm{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]. \end{aligned}$$

Dobili smo torej

$$v(s) = \mathrm{E}[R_{t+1} + \gamma v(S_{t+1}) \mid S_t = s]$$

oziroma

$$v(s) = \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s'), \tag{3}$$

kjer smo upoštevali aditivnost in idempotentnost pogojnega matematičnega upanja. Enačba (3) je **Bellmanova enačba za Markovske procese nagrajevanja**.

Ker so namizne igre primer, ko je stanj končno mnogo, torej n, jih kot v definiciji 2.8 označimo z $1, \ldots, n$. Bellmanovo enačbo nato lahko prepišemo v matrični obliki

$$\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathcal{R}_1 \\ \vdots \\ \mathcal{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathcal{P}_{11} \ldots \mathcal{P}_{1n} \\ \vdots \\ \mathcal{P}_{n1} \ldots \mathcal{P}_{nn} \end{bmatrix} \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$$

ali krajše

$$v = \mathcal{R} + \gamma \mathcal{P} v.$$

Opazimo, da je to *sistem linearnih enačb*, ki ima eksplicitno rešitev:

$$\begin{aligned} v &= \mathcal{R} + \gamma \mathcal{P} v \\ (I - \gamma \mathcal{P}) v &= \mathcal{R} \\ v &= (I - \gamma \mathcal{P})^{-1} \mathcal{R}. \end{aligned}$$

Ta rešitev predpostavlja obrnljivost matrike $(I - \gamma \mathcal{P})$ in računanje njenega inverza (oz. reševanje sistema), kar zahteva $O(n^3)$ operacij, zato je smiselna samo za majhne

procese. Dobra stran pa je, da nam obrnljivost matrike zagotavlja enolično rešitev sistema. Za večje procese obstajajo iterativni algoritmi in metode, nekatere izmed njih bomo spoznali v naslednjem odseku.

2.3.3. *Markovski proces odločanja.* Kot nakazuje že ime, **Markovski proces odločanja** (angl. *Markov decision process (MDP)*) razširja koncept procesa nagrajevanja z dodatkom odločanja – akcij. S tem dodatkom imamo v popolnosti opisan problem spodbujevalnega učenja: proučujemo torej nek proces, kjer ima agent možnost odločanja, izid pa je vsaj delno slučajen in odvisen od okolja. V nadaljevanju bomo Markovske procese odločanja označevali z angleško krajšavo, tj. MDP.

Opomba 2.10. Ker v Markovskem procesu odločanja nastopajo (in imajo poglavitno) vlogo akcije, seveda pomembno vplivajo na nagradno funkcijo in prehodno matriko, zato moramo ta koncepta ustrezno prilagoditi. Definiramo

$$\mathcal{R}_s^a = E[R_{t+1} \mid S_t = s, A_t = a],$$
$$\mathcal{P}_{ss'}^a = P(S_{t+1} = s' \mid S_t = s, A_t = a)$$

Definicija 2.11 (Markovski proces odločanja)**.** Naboru $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$, za katerega velja

- $\mathcal{S}$ je (končna) množica stanj,
- $\mathcal{A}$ je (končna) množica akcij,
- $\mathcal{P}$ je prehodna matrika $\mathcal{P}_{ss'}^a = P(S_{t+1} = s' \mid S_t = s, A_t = a)$,
- $\mathcal{R}$ je *nagradna funkcija* $\mathcal{R}_s^a = \mathrm{E}[R_{t+1} \mid S_t = s, A_t = a]$,
- $\gamma \in [0, 1]$ je *diskontni faktor*,

pravimo **Markovski proces odločanja** (angl. *Markov decision process*).

V kontekstu MDP-jev lahko sedaj formalno definiramo strategijo agenta π, ki je zahvaljujoč Markovski lastnosti odvisna od enega (trenutnega) stanja in ne od celotne zgodovine procesa. Definiramo tudi vrednostno funkcijo stanja $v_\pi(s)$ in vrednostno funkcijo akcije $q_\pi(s, a)$ kot v (1) in (2). Njihove definicije se torej ne spremenijo, so pa sedaj formalno umeščene v model.

Opazimo, da je v MDP-ju pri fiksni strategiji π zaporedje oz. proces stanj $S_1, S_2, \ldots$ Markovska veriga $(\mathcal{S}, \mathcal{P}^\pi)$. Če v zaporedje stanj pomešamo še nagrade, torej $S_1, R_2, S_2, R_3, \ldots$, dobimo Markovski proces nagrajevanja $(\mathcal{S}, \mathcal{P}^\pi, \mathcal{R}^\pi, \gamma)$, kjer so elementi matrike $\mathcal{P}^\pi$ in vektorja $\mathcal{R}^\pi$ enaki

$$\mathcal{P}_{ss'}^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{P}_{ss'}^a$$
$$\mathcal{R}_s^\pi = \sum_{a \in \mathcal{A}} \pi(a|s) \mathcal{R}_s^a.$$

Opomba 2.12. Morda je nekatere bralce zbodlo, da v zaporedju stanj in nagrad stanju S_1 ne sledi R_1, temveč R_2. Za tako notacijo smo se odločili, da poudarimo, da okolje agentu sočasno poda stanje S_2 in nagrado R_2, glede na stanje S_1 pa se agent odloča o akciji A_1.

MDP-ji so splošna orodja za obravnavo stohastičnih procesov, ki vključujejo odločitve v diskretnem času. Njihov utemeljitelj je Richard Bellman, ki je znan predvsem po izumu dinamičnega programiranja, zato morda ni presenetljivo, da nam prav dinamično programiranje poda osnovo za njihovo reševanje. Kot pri procesih

nagrajevanja, lahko tudi tu izpeljemo Bellmanovo enačbo. Najprej analogno kot prej izpeljemo zvezi

$$v_\pi(s) = \mathrm{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s],$$
$$q_\pi(s,a) = \mathrm{E}[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) \mid S_t = s, A_t = a].$$

Iz definicij $v_\pi(s)$ in $q_\pi(s,a)$ sledi, da če za določeno stanje s pri strategiji π seštejemo vse vrednosti $q_\pi(s,a)$ in to vsoto utežimo z verjetnostmi izbire akcije, dobimo ravno $v_\pi(s)$ oz.

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) q_\pi(s,a). \tag{4}$$

S podobnim razmislekom lahko dobimo še eno zvezo med tema dvema funkcijama. Če začnemo s parom stanja in akcije (s,a), nas potem od naslednjega stanja s' loči le še nagrada. Vrednost para je potem odvisna od te nagrade $\mathcal{R}_s^a$ in pa od vrednosti stanja s'. Ta vrednost je odvisna od tega, v katero stanje nas okolje pahne oz. od matrike $\mathcal{P}$ – gledati moramo pričakovano vrednost. Prav tako pa je stanje s' v časovnem smislu en korak kasneje, zato mora biti vrednost diskontirana z γ. Ta razmislek zapišemo kot

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s'). \tag{5}$$

Če enačbi (4) in (5) vstavimo drugo v prvo in nato prvo v drugo, dobimo

$$v_\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \right], \tag{6}$$

$$q_\pi(s,a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \left[\sum_{a' \in \mathcal{A}} \pi(a'|s') q_\pi(s',a') \right]. \tag{7}$$

Enačbo (6) imenujemo **Bellmanova enačba pričakovanja** (angl. *Bellman expectation equation*), ki jo lahko zapišemo v matrični obliki:

$$v_\pi = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_\pi.$$

Iz te enačbe lahko izrazimo v_π in dobimo enačbo

$$(I - \gamma \mathcal{P}^\pi) v_\pi = \mathcal{R}^\pi,$$

ki ima v primeru obstoja inverza matrike $I - \gamma \mathcal{P}^\pi$ enolično rešitev.

Ta enačba je sicer pomembna, a določi samo prave vrednosti glede na neko strategijo. Nas pa zanima »rešitev« MDP-ja. To pomeni, da želimo najti strategijo, ki nam bo prinesla največjo nagrado, tj. optimalno strategijo.

2.3.4. *Optimalne vrednostne funkcije.*

Definicija 2.13.

- **Optimalna vrednostna funkcija stanja** (angl. *optimal state-value function*) $v_*(s)$ je vrednostna funkcija stanja, kjer vedenje agenta določa strategija, ki vrne največje možne vrednosti

$$v_*(s) = \max_\pi v_\pi(s).$$

- **Optimalna vrednostna funkcija akcije** (angl. *optimal action-value function*) $q_*(s,a)$ je analog optimalni vrednostni funkciji stanja
$$q_*(s,a) = \max_\pi q_\pi(s,a).$$
- Na množici vseh možnih strategij Π definiramo delno urejenost na naslednji način:
$$\pi \geq \pi', \text{ če } v_\pi(s) \geq v_{\pi'}(s)\ \forall s.$$
Če za strategiji velja ta neenakost, pravimo, da je π *boljša ali enaka* kot π'.
- **Optimalna strategija** je strategija π_*, ki je boljša ali enaka od vseh ostalih strategij $\pi \in \Pi$.

Sedaj lahko uporabimo naslednji izrek.

Izrek 2.14. *Za vsak končni Markovski proces odločanja s končnimi nagradami velja:*

(1) *Vedno obstaja optimalna strategija π_* (ne nujno enolična), ki je boljša ali enaka kot vse ostale strategije; $\pi_* \geq \pi$ za vsak $\pi \in \Pi$.*
(2) *Vedno obstaja optimalna strategija, ki je deterministična. Obstaja torej funkcija $\pi_* : \mathcal{S} \to \mathcal{A}$, $s \mapsto \pi_*(s) = a$.*
(3) *Vse optimalne strategije določajo enako optimalno vrednostno funkcijo stanja in optimalno vrednostno funkcijo akcije:*
$$v_{\pi*}(s) = v_*(s),$$
$$q_{\pi*}(s,a) = q_*(s,a).$$

Izrek je dokazan v [12], za namene naloge pa je najbolj pomemben rezultat to, da ob poznani $q_*(s,a)$, poznamo tudi optimalno deterministično strategijo, ki jo dobimo kot $\pi_*(s) = \arg\max_{a\in\mathcal{A}} q_*(s,a)$. To strategijo lahko zapišemo tudi v stohastični obliki, in sicer $\pi_*(a|s) = \mathbb{1}(a = \arg\max_{a\in\mathcal{A}} q_*(s,a))$.

Podobno kot za poljubne vrednostne funkcije lahko tudi za optimalne vrednostne funkcije izpeljemo Bellmanovo enačbo, ki jo sedaj imenujemo **Bellmanova enačba optimalnosti** (angl. *Bellman optimality equation*). Natančna izpeljava je stvar dokaza zgornjega izreka, a bistvo je, da dobimo spodnji zvezi:
$$v_*(s) = \max_{a\in\mathcal{A}} q_*(s,a),$$
$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_*(s).$$
Ti zvezi nato vstavimo eno v drugo, in dobimo Bellamanovi enačbi:
$$v_*(s) = \max_{a\in\mathcal{A}}(\mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a v_*(s)),$$
$$q_*(s,a) = \mathcal{R}_s^a + \gamma \sum_{s'\in\mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'\in\mathcal{A}} q_*(s',a').$$
Opazimo, da sta enačbi tokrat nelinearni, kar močno oteži direktno reševanje. Zaradi tega se v praksi pri iskanju optimalnih strategij poslužujemo drugačnih metod.

3. Algoritmi pri spodbujevalnem učenju

Algoritmov za reševanje MDP-jev je precej. Mi se bomo omejili predvsem na algoritme, ki se učijo prek vrednostne funkcije in izhajajo iz dinamičnega programiranja, a naj na tem mestu omenimo, da obstajajo tudi algoritmi, ki posodabljajo strategijo neposredno. Opisujeta jih Sutton in Barto [11].

Izjemnega pomena pri reševanju je vprašanje, ali sta znani matrika $\mathcal{P}$ in nagradna funkcija $\mathcal{R}_s^a$. Izkaže se, da je večina problemov takih, da ti dve stvari nista znani. Eden izmed takih problemov je tudi igranje namiznih iger – ne poznamo strategije našega nasprotnika, zato ne poznamo verjetnosti prehodov med stanji, poleg tega pa ima lahko tudi igra sama element stohastičnosti (npr. met kocke).

Naj na tem mestu omenimo, da reševanju celotnega problema spodbujevalnega učenja pravimo **načrtovanje** (angl. *planning*), le-ta pa se deli na dve stopnji:

- **Napovedovanje** - pri tem podproblemu podamo nek MDP in strategijo, algoritem nam vrne vrednostno funkcijo stanja (glede na strategijo).
- **Upravljanje** - to je bolj poln problem: podan je MDP, algoritem pa nam vrne optimalno vrednostno funkcijo in pripadajočo optimalno strategijo.

3.1. **Dinamično programiranje – reševanje poznanih MDP-jev.** Dinamično programiranje je optimizacijska metoda, ki deluje na principu deljenja velikega problema na manjše prekrivajoče podprobleme. V jedru reševanja je **Bellmanova enačba**, ki opisuje odnos med vrednostmi podproblemov in vrednostjo glavnega problema. Ker je idejo dinamičnega programiranja in MDP-jev dobil Bellman ob istem času, je naravno, da je dinamično programiranje metoda, ki je prilagojena prav situaciji v MDP-ju.

Problem mora v splošnem zadoščati dvema lastnostima, da je zanj primerno reševanje z dinamičnim programiranjem. Prvi pravimo **lastnost optimalne podstrukture**, ki pravi, da lahko problem razstavimo na podprobleme in rešitve podproblemov lahko potem sestavimo nazaj v rešitev celotnega problema. Druga pomembna lastnost pa so **prekrivajoči podproblemi**, ki implicira, da lahko že poznane rešitve podproblemov večkrat uporabimo.

Opazimo, da obe lastnosti veljata za MDP-je; Bellmanova enačba nam razdeli problem na podprobleme, vrednostna funkcija pa hrani in ponovno uporablja rešitve. Lastnosti veljata tudi za Markovske procese nagrajevanja, zato lahko metode, ki jih bomo spoznali, uporabimo za reševanje tovrstnih procesov. Nam je seveda v interesu reševati MDP-je, zato se bomo posvetili temu.

3.1.1. *Iterativno ocenjevanje strategije.* Algoritem deluje na MDP-ju, katerega prehodna matrika je znan podatek. Poleg tega potrebuje tudi fiksno strategijo π. Algoritem torej rešuje zgolj problem napovedovanja. Glavna ideja je uporaba Bellmanove enačbe pričakovanja (6). Z njo na vsakem koraku posodobimo $v_k(s)$ v $v_{k+1}(s)$ s pomočjo podatka $v(s')$ za vsa stanja s', ki sledijo stanju s. Enačba iteracije se potem glasi

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \left[\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right]$$

ali na kratko

$$v_{k+1} = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v_k.$$

Izrek 3.1. *Iterativno ocenjevanje strategije konvergira proti vrednostni funkciji, ki jo določa strategija π.*

Izrek bomo dokazali samo v primeru končnih MDP-jev, to pomeni, da je prostor stanj $\mathcal{S}$ končen, torej $|\mathcal{S}| < \infty$. V tem primeru je prostor vrednostnih funkcij $\mathcal{V}$ vektorski prostor. Vsaka točka v njem korespondira z eno vrednostno funkcijo $s \mapsto v(s)$.

Definicija 3.2. Neskončna norma za vrednostne funkcije je norma, ki meri »razdaljo« med vrednostnima funkcijama v in u. Izračunamo jo kot največjo razliko med vrednosti stanj oz.

$$||v - u||_\infty = \max_{s \in \mathcal{S}} |v(s) - u(s)|.$$

Dokaz izreka 3.1. Pokazali bomo, da iz iteracije lahko pridobimo skrčitev, iz Banachovega izreka o fiksni točki potem sledi naš rezultat. Definiramo **Bellmanov operator pričakovanja** kot

$$T_\pi(v) = \mathcal{R}^\pi + \gamma \mathcal{P}^\pi v.$$

Naj bosta u in v vrednostni funkciji. Potem je zgornji operator skrčitev glede na neskončno normo:

$$\begin{aligned} ||T_\pi(v) - T_\pi(u)||_\infty &= ||\mathcal{R}^\pi + \gamma \mathcal{P}^\pi v - \mathcal{R}^\pi + \gamma \mathcal{P}^\pi u||_\infty \\ &= ||\gamma \mathcal{P}^\pi (v - u)||_\infty \\ &\leq ||\gamma \mathcal{P}^\pi||_\infty ||v - u||_\infty \\ &\leq \gamma ||v - u||_\infty, \end{aligned}$$

kjer smo upoštevali, da velja $||\mathcal{P}^\pi||_\infty \leq 1$, saj so elementi $\mathcal{P}^\pi$ verjetnosti. Po Bellmanovi enačbi pričakovanja je v_π fiksna točka za T_π, rezultat nato sledi iz Banachovega skrčitvenega izreka. □

S tem procesom tako res dobimo vrednostno funkcijo, ki jo določa strategija, a strategija je statična; to reši problem napovedovanja.

Da rešimo problem upravljanja, uporabimo enostavno idejo: strategijo izboljšamo **požrešno**. To pomeni, da izberemo tako akcijo, ki nas pripelje v stanje s, ki je dosegljivo in ima največjo vrednost $v_\pi(s)$. Na vsakem koraku torej izberemo največjo vrednost ne glede na vse, od tu pride ime *požrešno*.

3.1.2. *Iteracija strategije.* Algoritem, ki ga dobimo s tem, da združimo iterativno ocenjevanje strategije in požrešno izboljšavo strategije, imenujemo iteracija strategije (angl. *policy iteration*). S tem algoritmom lahko sedaj rešimo problem upravljanja in s tem celoten problem določitve rešitve MDP-ja. V praksi to pomeni, da začnemo z naključno strategijo, jo iterativno ocenimo, nato na podlagi rezultata požrešno izboljšamo strategijo in ta dva koraka ponavljamo.

Izrek 3.3. *Iteracija strategije vedno konvergira k optimalni strategiji π_*.*

Dokaz. Ukvarjali se bomo samo z determinističnimi strategijami, saj so zaradi požrešne izbire strategije le-te vedno deterministične, razen začetne. Ker je izbira začetne strategije za trditev nepomembna, lahko izberemo neko naključno deterministično strategijo $a = \pi(s)$.

Požrešno vedenje zdaj za nas pomeni, da vzamemo

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} q_\pi(s, a). \tag{8}$$

To očitno izboljša vrednost za vsako stanje:

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) \geq q_\pi(s, \pi(s)) = v_\pi(s).$$

Posledično izboljša tudi vrednostno funkcijo:

$$\begin{aligned} v_\pi(s) &\leq q_\pi(s, \pi'(s)) = \mathrm{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\ &\leq \mathrm{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) \mid S_t = s] \\ &\leq \mathrm{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 q_\pi(S_{t+2}, \pi'(S_{t+2})) \mid S_t = s] \\ &\leq \mathrm{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \ldots \mid S_t = s] = v_{\pi'}(s). \end{aligned}$$

Če vrednost preneha rasti, dobimo

$$q_\pi(s, \pi'(s)) = \max_{a \in \mathcal{A}} q_\pi(s, a) = q_\pi(s, \pi(s)) = v_\pi(s).$$

To se zgodi, saj zaradi končnih vrednosti nagrad tudi vrednostne funkcije zavzemajo omejene vrednosti. Potem opazimo, da je zaradi $v_\pi(s) = \max_{a \in \mathcal{A}} q_\pi(s, a)$ zadoščeno Bellmanovi enačbi optimalnosti in zato po izreku 2.14 velja $v_\pi(s) = v_*(s)$ za vse $s \in \mathcal{S}$, torej je π optimalna strategija. □

3.1.3. *Iteracija vrednosti.* K problemu lahko pristopimo tudi neposredno prek Bellmanove enačbe optimalnosti in se s tem izognemo neposredni uporabi strategije, vseeno pa rešimo problem upravljanja. Ideja je podobna kot pri iterativnem ocenjevanju strategije, le da imamo drugačno enačbo za iteracijo:

$$v_{k+1}(s) = \max_{a \in \mathcal{A}} (\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s'))$$

oziroma

$$v_{k+1} = \max_{a \in \mathcal{A}} (\mathcal{R}^a + \gamma \mathcal{P}^a v_k).$$

Podobno kot pri iterativnem ocenjevanju strategije pokažemo, da zgornja iteracija konvergira k v_*.

3.1.4. *Časovna zahtevnost.* Za korak obeh iteracij je potrebnih $O(mn^2)$ operacij, kjer je m število akcij in n število stanj. Popolnoma enake algoritme lahko uporabimo za iteracijo $q_\pi(s, a)$ oz. $q_*(s, a)$, le da pridemo v tem primeru do časovne zahtevnosti $O(m^2n^2)$.

3.2. **Monte Carlo.** Do sedaj smo se ukvarjali s poznanim MDP-jem, kar pa ni naš končen cilj. V nadaljevanju se bomo posvetili nepoznanim. Metode, ki jih bomo spoznali so tako bolj splošno aplikativne, svoje bistvo pa črpajo pri do sedaj opisanem.

Prva taka metoda je Monte Carlo. Spopada se s problemom *napovedovanja* z dodatno izboljšavo, da ne rabi poznati matrike $\mathcal{P}$. Potrebujemo pa strategijo π, na podlagi katere opazujemo epizode dogajanja. Ideja delovanja je, da se namesto na direktno računanje pričakovanega povračila $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$, osredotočimo na računanje *empiričnega* povračila. To storimo tako, da za vsako stanje s definiramo števec $N(s)$, ki beleži, kolikokrat je bilo stanje obiskano. Ta števec ne beleži zgolj obiskov znotraj epizode, temveč skozi celoten proces učenja. Za vsako stanje hranimo še $S(s)$, ki ga posodobimo za vsa stanja na koncu epizode tako, da prištejemo dejansko opaženo empirično povračilo G_t, ki pripada posameznemu stanju. Po koncu učenja enostavno za vsako stanje delimo $S(s)$ z $N(s)$ in tako dobimo vzorčno povprečje za dejanski $v(s)$. Povzemimo opisano:

- Ob vsakem času t, ko je obiskano stanje s ($S_t = s$):

$$N(s) \leftarrow N(s) + 1,$$
$$S(s) \leftarrow S(s) + G_t.$$

- Po zadostni količini epizod (ob koncu učenja):

$$V(s) \leftarrow S(s)/N(s).$$

Opomba 3.4. V izrazu $N(s) \leftarrow N(s) + 1$ (in v podobnih izrazih v nadaljevanju) operator »←« pomeni prirejanje vrednosti na desni strani spremenljivki na levi, torej *spremenljivka* ← *vrednost*.

Opazimo, da metoda deluje samo za probleme, ki imajo končne epizode, saj šele takrat lahko poznamo G_t. Če za vsako stanje s hranimo $N(s)$ in $S(s)$, tako dobimo oceno za $v_\pi(s)$ za vsak $s \in \mathcal{S}$ pri dani strategiji π.

Ker je v ozadju našega problema računanje približka pričakovane vrednosti s pomočjo povprečja nekega vzorca, nam skoraj gotovo konvergenco $V(s)$ proti $v_\pi(s)$, ko gre $N(s)$ proti neskončno, zagotavlja krepki zakon velikih števil. To je res, ker je vsako opaženo povračilo dobljeno na podlagi enakih pogojev (strategije in okolja), torej je vzorec enako porazdeljen. Prav tako so vzorčna povračila med seboj neodvisna (ena epizoda ne vpliva na drugo) in imajo končno varianco (zaradi končnih nagrad). Na osnovi krepkega zakona velikih števil lahko ugotovimo, da je konvergenca algoritma *kvadratična*. To sta pokazala Sutton in Singh [9].

Prostorsko zahtevnost algoritma lahko izboljšamo, če upoštevamo, da povprečje μ nekega zaporedja slučajnih spremenljivk $X_1, X_2, \ldots$ izračunamo inkrementalno:

$$\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{i=1}^{k} X_i \\
&= \frac{1}{k}\left[X_k + \sum_{i=1}^{k-1} X_i\right] \\
&= \frac{1}{k}\left[X_k + (k-1)\mu_{k-1}\right] \\
&= \mu_{k-1} + \frac{1}{k}\left(X_k - \mu_{k-1}\right).
\end{aligned}$$

Na podlagi te zveze lahko implementiramo **inkrementalni Monte Carlo** algoritem. V našem primeru opazujemo zaporedje opaženih povračil za posamezno stanje tekom več epizod, inkrementalno pa računamo $V(S_t)$ za vsak t. Algoritem se tako glasi:

$$N(S_t) \leftarrow N(S_t) + 1,$$
$$V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t)).$$

Izkaže se, da lahko zgornje še malo posplošimo in namesto faktorja $1/N(S_t)$ uporabimo splošni faktor α

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t)), \tag{9}$$

kjer α imenujemo **velikost koraka** oz. **hitrost učenja** (angl. *learning rate*). Posodobitveno pravilo (9) je konkreten primer bistvene ideje spodbujevalnega učenja

prek vrednostne funkcije. Vsi ostali algoritmi nam dajo pravilo podobne oblike:

$$\textit{nova ocena} \leftarrow \textit{stara ocena} + \textit{korak}\,(\textit{tarča} - \textit{stara ocena}). \tag{10}$$

3.3. **Algoritem TD(0).** V spodbujevalnem učenju obstaja skupina algoritmov, ki se imenujejo **algoritmi učenja s časovno razliko** (angl. *temporal-difference learning*). Delujejo v podobnih pogojih kot Monte Carlo in tudi dosegajo enak cilj. Pomembna razlika je, da ne potrebujejo empiričnega povračila pri posodobitvi. Zaradi tega lahko delujejo tudi v sistemih, ki se ne delijo na epizode oz. ne potrebujejo končanih epizod za svoje delovanje.

Najpreprostejši algoritem učenja s časovno razliko je TD(0). Ideja je, da za vse čase t ocenimo povračilo kot $G_t \approx R_{t+1} + \gamma V(S_{t+1})$. To potem uporabimo za *tarčo* v pravilu (10). To nam da pravilo

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)). \tag{11}$$

Iz tega zelo hitro ugotovimo, od kod pride poimenovanje. Algoritem za svoje posodabljanje uporablja podatek iz naslednjega trenutka – torej se uči s časovno razliko. Pomembno je omeniti tudi, da za učenje uporablja oceno vrednosti naslednjega stanja, torej ocenjuje na podlagi ocene. Temu pravimo *zankanje* (angl. *bootstrapping*).

Seveda za delovanje ni nujno, da se agent uči med samo epizodo, ampak se lahko samo ob koncu – tako kot pri Monte Carlu. Mi se bomo zaradi enostavnosti implementacije omejili na ta primer. Ker so si algoritmi med sabo podobni, zadošča navesti zgled algoritma za TD(0).

Algorithm 1 TD(0) - ocenjevanje $V \approx v_\pi$

Podatki: strategija π, parameter hitrosti učenja α, število epizod $stEpizod$, diskontni faktor γ

Poljubno nastavimo vrednosti $V(s)$ za vsak $s \in \mathcal{S}$ (npr. $V(s) = 0$ za vsak $s \in \mathcal{S}$)

for $k = 1, 2, \ldots, stEpizod$ **do**
  Generiraj epizodo prek strategije π
  $stanja \leftarrow$ seznam vseh opaženih stanj
  $nagrade \leftarrow$ seznam vseh opaženih nagrad

  for $t = 1, 2, \ldots, length(stanja)$ **do**
    $trenutno = stanja[t]$
    $naslednje = stanja[t+1]$
    $V(trenutno) \leftarrow V(trenutno) + \alpha(nagrade[naslednje] + \gamma V(naslednje) - V(trenutno))$
  end for
end for

3.3.1. *Primerjava z Monte Carlo (povzeto po* [6]*).*

- Tarča pri MC je nepristranska ocena za $v_\pi(S_t)$, medtem ko je pri TD(0) tarča pristranska ocena. Ta prednost MC je uravnotežena s tem, da ima tarča pri TD(0) nižjo varianco, saj je odvisna od ene same slučajne spremenljivke, medtem ko je pri MC odvisna od mnogih slučajnih dogodkov.

- Prav tako je pomembna razlika v dejstvu, da MC nujno rabi popolne epizode in se lahko izvede le po končani epizodi, TD(0) pa se lahko uči sproti (angl. *online*) in sploh ne potrebuje končnih epizod. Seveda pa lahko TD(0) uporabimo tudi za učenje samo ob koncu posameznih epizod (angl. *offline*).
- Razlika je tudi v tem, kam konvergirata algoritma, če imamo na voljo samo končno količino epizod. Naj K označuje število epizod in naj k nakazuje, v kateri epizodi smo. Potem MC konvergira k rešitvi z minimalno povprečno kvadratno napako (MSE)

$$\sum_{k=1}^{K}\sum_{t=1}^{T_k}(G_t^k - V(s_t^k))^2,$$

  torej se najbolje prilega opazovanim povračilom. TD(0) pa konvergira k rešitvi modela največjega verjetja Markova; to je rešitev MDP-ja, ki se najbolje prilega podatkom. Dobimo torej naslednji cenilki, kjer $N(s,a)$ šteje, kolikokrat je bila izvedena katera akcija v stanju:

$$\hat{\mathcal{P}}_{ss'}^a = \frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{t=1}^{T_k}\mathbb{1}(s_t^k, a_t^k, s_{t+1}^k = s, a, s'),$$

$$\hat{\mathcal{R}}_s^a = \frac{1}{N(s,a)}\sum_{k=1}^{K}\sum_{t=1}^{T_k}\mathbb{1}(s_t^k, a_t^k = s, a)r_t^k.$$

- Zadnja pomembna razlika je, da TD izrabi Markovsko lastnost, medtem ko je MC ne. Zato je vsaj v teoriji TD veliko bolj učinkovit v MDP-jih.

Zgled 3.5 (Razlike med MC in TD – »ti si sodnik« [11])**.** *Recimo, da imamo na voljo za učenje naslednjih 8 epizod:*

- $A \to 0 \to B \to 0$,
- $B \to 1$,
- $B \to 1$,
- $B \to 1$,
- $B \to 1$,
- $B \to 1$,
- $B \to 1$,
- $B \to 0$.

Te oznake pomenijo, da smo prvo epizodo začeli v stanju A, nato dobili nagrado 0 in prešli v stanje B, kjer smo ponovno dobili nagrado 0, tu pa se je epizoda končala. Podobno za ostale epizode. Če bi se učili samo iz teh osmih epizod, bi tako MC kot TD(0) ocenila $V(B) = 3/4$, saj je v šestih opažanjih B dobil nagrado 1, v dveh pa 0.

Razlike se vidijo pri stanju A. Ker MC algoritem minimizira povprečno kvadratno napako, po opaženem stanju A pa je bila nagrada vedno (torej enkrat) enaka 0, zato ta algoritem dodeli vrednost $V(A) = 0$.

TD(0) pa pristopi nekoliko drugače. Vedno, ko je bilo opaženo stanje A, je bilo potem takoj opaženo stanje B (vmes smo prejeli nagrado 0), zato je smiselno, da je vrednost stanja A enaka vrednosti stanja B, tj. $V(A) = 3/4$. Pri tem pristopu je bil problem modeliran kot markovski proces.

3.4. **Algoritem TD(λ).** V osnovi obeh algoritmov, ki smo ju spoznali, je pravilo (10). Izkaže pa se, da sta oba samo ekstremna primera algoritma, ki ga bomo spoznali sedaj. V ta namen si poglejmo povračila, ki gledajo n-korakov v prihodnost. Tarča se primerno prilagodi v

$$G_t^{(n)} = R_{t+1} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n V(S_{t+1}).$$

To tarčo lahko seveda uporabimo v pravilu (10). S tem dobimo algoritem, ki se imenuje *učenje s časovno razliko z n koraki*. Opazimo tudi, da če pošljemo n proti neskončno, pridemo do že znanega MC pravila. Hitro tudi vidimo, da to pravilo sicer res poveže TD(0) in MC, a ne predstavlja nobene bistvene izboljšave. Pojavi pa se še dodaten problem. Če smo v času, ki je manj kot n korakov stran od konca epizode, ne moremo porabiti celotnega $G_t^{(n)}$. V tem primeru enostavno vzamemo toliko korakov naprej, kot nam čas dopušča.

Dejansko nadgradnjo algoritma nam da dejstvo, da lahko skupaj združujemo podatke iz različnih časov prek tega, da povprečimo $G_t^{(n)}$ za različne n. V nadaljevanju bomo pogledali, kako združimo podatke iz *vseh* časov, ki nastopijo. Najti moramo ustrezne uteži, ki se seštejejo v ena in so smiselne glede na obravnavan problem. Primernega kandidata za tarčo je leta 1988 našel Richard S. Sutton [10], in sicer:

$$G_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{(n-1)} G_t^{(n)},$$

kjer je $\lambda \in [0,1]$ poljubna. Od tu takoj sledi algoritem **TD(λ) s pogledom naprej** (angl. *forward-view TD(λ)*). V njegovi osnovi je pravilo

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t^\lambda - V(S_t)).$$

Poimenovanje izhaja iz dejstva, da kot MC tudi ta algoritem potrebuje znanje prihodnosti, da posodobi vrednosti; z drugimi besedami, uči se lahko samo iz popolnih epizod.

Čeprav TD(λ) je teoretično zanimiv algoritem, je v praksi veliko bolj uporaben algoritem, ki ne potrebuje vedenja o prihodnosti in se tako lahko uči »*online*«. Izkaže se, da je zgornji algoritem mogoče popraviti in ga spremeniti v bolj praktično verzijo.

3.4.1. *Sledi upravičenosti (angl. Eligibility traces).* Preko tega koncepta spremenimo algoritem tako, da se lahko uči po vsaki potezi sproti. Ideja je, da za vsako stanje hranimo njegovo sled upravičenosti. Konceptualno to pomeni, da sproti ugotavljamo, kako zasluženo je stanje za morebitno nagrado – koliko je pripomoglo k njeni pridobitvi.

Praktično pa je sled upravičenosti preslikava $E_t : \mathcal{S} \to \mathbb{R}^+$. Pri sledeh sta pomembni dve stvari: **čas**, ki je pretekel od obiska, in **pogostost** obiska stanja. Sledi upravičenosti združijo oboje. Za neko stanje s tako lahko napišemo enostavno enačbo, kjer je z $E_t(s)$ označena sled v času t za stanje s:

$$\begin{aligned} E_0(s) &= 0, \\ E_t(s) &= \gamma\lambda E_{t-1}(s) + \mathbb{1}(S_t = s), \end{aligned}$$

pri čemer sta γ in λ že poznana parametra. Pomensko torej sled izginja z oddaljevanjem od obiska s, v primeru, da je s ponovno obiskan, pa skoči za 1. S tem lahko torej dodelimo nagrade oz. kazni samo tistim stanjem, ki to zaslužijo. Hkrati pa tudi ne vrednotimo enako vseh obiskanih stanj – pomemben je čas.

S tem novim konceptom lahko posodobimo TD(λ). Če označimo popravek algoritma TD(0) z $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$, potem zapišemo

$$V(s) \leftarrow V(s) + \alpha\delta_t E_t(s). \tag{12}$$

Algoritem TD(λ) se s tem prilagodi v **TD(λ) s pogledom nazaj** (angl. *backward-view TD(λ)*). Strukturno je praktično enak algoritmu TD(0), le da je popravek pomnožen še s sledjo upravičenosti.

Opomba 3.6. Poleg zgoraj definiranih sledi, ki jim pravimo *akumulativne sledi*, obstajata še dve alternativi:

- *Nadomeščujoče sledi*, ki so podobne akumulativnim, le da ob obisku ni skoka za 1, pač pa skok na 1, drugače pa imajo enako lastnost propadanja kot akumulativne.
- *Nizozemske sledi*, ki jih definiramo kot $E_t(s) = (1-\alpha)\gamma\lambda E_{t-1}(s) + \mathbb{1}(S_t = s)$. Torej podobno kot akumulativne, le da so dodatno pomnožene z $(1-\alpha)$. Ideja je, da s tem dodatnim členom dobimo sled, ki je med akumulativno in nadomeščujočo.

Mi bomo uporabljali prvotno definicijo sledi upravičenosti, torej akumulativne sledi.

V formulaciji pravila TD(λ) s pogledom nazaj postane očitno, zakaj ta algoritem poveže TD(0) in MC:

- Če je $\lambda = 0$, potem velja $E_t(s) = \mathbb{1}(S_t = s)$, torej se posodobi samo trenutno stanje: $V(S_t) \leftarrow V(S_t) + \alpha\delta_t$, kar pa je natančno posodobitev pri TD(0).
- Pri $TD(1)$ pa je za nagrado zaslužno vsako stanje, tako kot pri MC.

Pravo ekvivalenco nam pomaga videti naslednji izrek.

Izrek 3.7. *Vsota posodobitev vrednosti ob koncu epizode je enaka za TD(λ) s pogledom nazaj in pogledom naprej:*

$$\sum_{t=1}^{T} \alpha\delta_t E_t(s) = \sum_{t=1}^{T} \alpha \left(G_t^\lambda - V(S_t)\right) \mathbb{1}(S_t = s).$$

Dokaz. Oglejmo si napako pri TD(λ) s pogledom naprej:

$$\begin{aligned}
G_t^\lambda - V(S_t) &= -V(S_t) + (1-\lambda)\sum_{n=1}^{\infty} \lambda^{(n-1)} G_t^{(n)} \\
&= -V(S_t) \\
&+ (1-\lambda)\lambda^0(R_{t+1} + \gamma V(S_{t+1})) \\
&+ (1-\lambda)\lambda^1(R_{t+1} + \gamma R_{t+2} + \gamma^2 V(S_{t+2})) \\
&+ (1-\lambda)\lambda^2(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 V(S_{t+3})) \\
&+ \dots \\
&= -V(S_t) \\
&+ R_{t+1} + \gamma V(S_{t+1}) - \lambda R_{t+1} - \gamma\lambda V(S_{t+1}) \\
&+ \lambda R_{t+1} + \gamma\lambda R_{t+2} + \gamma^2\lambda V(S_{t+2}) - \lambda^2 R_{t+1} - \gamma\lambda^2 R_{t+2} - \gamma^2\lambda^2 V(S_{t+2}) \\
&+ [\lambda^2 R_{t+1} + \gamma\lambda^2 R_{t+2} + \gamma^2\lambda^2 R_{t+3} + \gamma^3\lambda^2 V(S_{t+3}) \\
&- \lambda^3 R_{t+1} - \gamma\lambda^3 R_{t+2} - \gamma^2\lambda^3 R_{t+3} - \gamma^3\lambda^3 V(S_{t+3})] \\
&+ \dots \\
&= (\gamma\lambda)^0(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \\
&+ (\gamma\lambda)^1(R_{t+2} + \gamma V(S_{t+2}) - V(S_{t+1})) \\
&+ (\gamma\lambda)^2(R_{t+3} + \gamma V(S_{t+3}) - V(S_{t+2})) \\
&+ \dots \\
&= \delta_t + \gamma\lambda\delta_{t+1} + (\gamma\lambda)^2\delta_{t+2} + \dots
\end{aligned}$$

Oglejmo si sedaj epizodo, kjer je stanje s obiskano natanko enkrat, in sicer ob času k. Sled upravičenosti je potem enaka

$$E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbb{1}(S_t = s) = (\gamma\lambda)^{t-k}\mathbb{1}(t \geq k).$$

Skozi celotno epizodo se napaka posodablja kot

$$\sum_{t=1}^{T} \alpha\delta_t E_t(s) = \alpha \sum_{t=k}^{T} (\gamma\lambda)^{t-k}\delta_t = \alpha(G_k^\lambda - V(S_k)),$$

kar pa je ravno enako kot TD(λ) s pogledom naprej ob isti predpostavki, da je stanje s obiskano samo ob času k. Če je stanje obiskano večkrat, pa se napaka akumulira v več takih napak pogleda naprej. □

3.5. **Problem upravljanja.** Do sedaj smo se osredotočali samo na problem napovedovanja – o strategiji nismo veliko govorili, smo pa ugotovili, kako ob dani strategiji dobimo pravo funkcijo vrednosti. Pri dinamičnem programiranju smo strategijo izboljšali s pomočjo požrešne izboljšave strategije, kar v grobem pomeni, da s pomočjo funkcije q_π v vsakem stanju izberemo akcijo, ki ima največjo vrednost. To smo opisali z enačbo (8) oz.

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} q_\pi(s, a).$$

Ta koncept bi lahko uporabili tudi v drugih do sedaj predstavljenih algoritmih. če bi v enačbah zamenjali funkcijo v za funkcijo q. Tako bi rešili celoten problem upravljanja, torej problem spodbujevalnega učenja. Naletimo pa na novo težavo. Za razliko od dinamičnega programiranja v tem primeru ne poznamo okolja, pač pa imamo do njega dostop le prek naključnih epizod. Če se vedemo požrešno, ni nujno, da raziščemo celoten prostor stanj $\mathcal{S}$. Prišli smo do problema *raziskovanja in izkoriščanja*. Na srečo pa ima problem enostavno rešitev.

Če se agent ne vede popolnoma požrešno, pač pa z verjetnostjo $\epsilon \in [0, 1]$ izbere naključno akcijo, je zagotovljeno, da bo z neničelno verjetnostjo obiskal vsa stanja, hkrati pa se bo strategija vseeno izboljševala, če bo le ϵ dovolj majhen. To nas pripelje do koncepta **ϵ-požrešnega raziskovanja**, ki deterministično strategijo spremeni v stohastično. Pri tem verjetnosti izbire akcije a v stanju s definiramo kot

$$\pi'(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{če } a = \arg\max_{a \in \mathcal{A}} q_\pi(s, a), \\ \epsilon/m & \text{sicer,} \end{cases} \tag{13}$$

pri čemer m označuje število akcij, ki so na voljo v stanju s.

Izrek 3.8. *Za vsako ϵ-požrešno strategijo π, je ϵ-požrešna strategija π', pridobljena glede na q_π s formulo* (13)*, izboljšava. Torej $v_{\pi'}(s) \geq v_\pi(s)$ za vsa stanja $s \in \mathcal{S}$.*

Dokaz. Če nam uspe pokazati, da $q_\pi(s, e) \geq v_\pi(s)$, kjer je e akcija pridobljena z ϵ-požrešno strategijo π', potem lahko analogno kot v dokazu izreka 3.3 pokažemo,

da velja $v_{\pi'}(s) \geq v_\pi(s)$. Prva neenakost sledi iz

$$\begin{aligned} q_\pi(s,e) &= \sum_{a\in\mathcal{A}} \pi'(a|s) q_\pi(s,a) \\ &= \frac{\epsilon}{m} \sum_{a\in\mathcal{A}} q_\pi(s,a) + (1-\epsilon) \max_{a\in\mathcal{A}} q_\pi(s,a) \\ &\geq \frac{\epsilon}{m} \sum_{a\in\mathcal{A}} q_\pi(s,a) + (1-\epsilon) \sum_{a\in\mathcal{A}} \frac{\pi(a|s) - \epsilon/m}{1-\epsilon} q_\pi(s,a) \\ &= \sum_{a\in\mathcal{A}} \pi(a|s) q_\pi(s,a) = v_\pi(s) \end{aligned}$$

□

Z upoštevanjem ϵ-požrešnega raziskovanja lahko sedaj zapišemo, kako bi s pomočjo algoritma TD(0) rešili celoten problem spodbujevalnega učenja. Postopek je podan v algoritmu 2.

Algorithm 2 TD(0) - ocenjevanje $Q \approx q_*$ (t. i. *SARSA* algoritem)

Podatki: parameter hitrosti učenja α, število epizod *stEpizod*, diskontni faktor γ, parameter požrešnosti ϵ

Poljubno nastavimo vrednosti $Q(s,a)$ za vsak $s \in \mathcal{S}$ in $a \in \mathcal{A}$ (npr. $Q(s,a) = 0$)

for $k = 1, 2, \ldots, stEpizod$ **do**
  Generiraj epizodo prek funkcije Q ϵ-požrešno
  $stanja \leftarrow$ seznam vseh opaženih stanj
  $nagrade \leftarrow$ seznam vseh opaženih nagrad
  $akcije \leftarrow$ seznam vseh opaženih akcij

  for $t = 1, 2, \ldots, length(stanja)$ **do**
    $s = stanja[t]$
    $s' = stanja[t+1]$
    $a = akcije[t]$
    $a' = akcije[t+1]$
    $Q(s,a) \leftarrow Q(s,a) + \alpha(nagrade[s'] + \gamma Q(s',a') - Q(s,a))$
  end for
end for

3.5.1. *Konvergenca.* Če ϵ-požrešno raziskovanje združimo z algoritmom TD(0), dobimo znan algoritem SARSA. Algoritem SARSA in podobni algoritmi, ki jih pridobimo z zamenjavo pravila TD(0) z drugimi, ki smo jih spoznali, je seveda uporaben samo, če konvergira. K sreči so konvergenco dokazali Singh idr. [8]. Mi bomo tu samo navedli potrebne definicije in izrek.

Definicija 3.9 (PLNR)**.** Zaporedje strategij $(\pi_k)_{k=1}^{\infty}$ je **požrešno v limiti z neskončnim raziskovanjem (PLNR)** (angl. *Greedy in the limit with infinite exploration*), če velja:

- Vsi pari stanj in akcij so obiskani oz. uporabljeni neskončnokrat.

- Zaporedje konvergira proti požrešni strategiji, torej

$$\lim_{k\to\infty} \pi_k(a|s) = \mathbb{1}(a = \arg\max_{a\in\mathcal{A}} q(s,a)).$$

Zgled 3.10. *Če imamo zaporedje ϵ-požrešnih strategij, je to zaporedje PLNR, če ϵ manjšamo proti* 0 *na vsakem koraku, recimo $\epsilon_k = 1/k$.*

Izrek 3.11. *Algoritem SARSA konvergira proti optimalni vrednostni funkciji akcij q_*, če veljata naslednja pogoja:*

- *Zaporedje strategij je PLNR (npr. ϵ-požrešno z $\epsilon_k = 1/k$).*
- *Zaporedje parametrov hitrosti učenja α_k zadošča pogojema*

$$\sum_{k=1}^{\infty} \alpha_k = \infty \quad in \quad \sum_{k=1}^{\infty} \alpha_k^2 < \infty.$$

Čeprav so za dokaz konvergence zgornji pogoji nujni, se je v praksi velikokrat izkazalo, da je možno doseči konvergenco tudi s fiksnima ϵ in α. Za tovrsten pristop smo se odločili tudi mi v praktičnem delu.

3.6. **Funkcijska aproksimacija.** Glavni problem opisanih algoritmov je shranjevanje v tabele. Vsi delujejo na principu grajenja velikih tabel; en vnos za vsako stanje (oz. en vnos za vsako kombinacijo stanja in akcije). To pri velikih MDP-jih ne pride v poštev, saj učenje postane zelo prostorsko in časovno zahtevno. Stanje, ki še ni bilo obiskano, nima dodeljene vrednosti, torej o njem ne vemo nič. Tabelarne rešitve zato potrebujejo veliko količino podatkov, torej veliko količino epizod.

Zgled 3.12 (Število stanj pri nekaterih namiznih igrah)**.** *Igra križci in krožci je zelo enostavna in to se odraža tudi v številu stanj v igri. Grobo oceno lahko dobimo, če upoštevamo, da lahko vsako od devetih polj vsebuje enega od znakov ali pa je prazno. To nam da oceno* $3^9 = 19683$*. Številko hitro zmanjšamo, če ne upoštevamo ilegalnih stanj. Tako dobimo* 5478 *legalnih stanj. Ob upoštevanju rotacij in zrcaljenj pa prispemo do* 765 *dejansko različnih stanj. Vsa ta stanja lahko enostavno shranimo v tabelo.*

Popolnoma druga zgodba pa nastopi pri igrah kot sta šah in go. Za šah je ocenjeno, da ima 10^{46} *stanj, go pa ima okoli* 10^{170} *stanj.*

Zato je pomembna nadgradnja algoritmov uporaba funkcijskih aproksimantov. Ta problem je izčrpno preučevan na področju numerike, statistike in nadzorovanega učenja in načeloma se lahko uporabi katerkoli od znanih aproksimantov. Ideja je, da z malim številom koeficientov/uteži w opišemo funkcijo, nato pa prilagajamo te koeficiente/uteži namesto funkcije neposredno. To zmanjša prostorsko zahtevnost algoritmov in pa tudi omogoča, da dodelimo neke (bolj ali manj smiselne) vrednosti stanjem, ki še niso bila obiskana. Ocene za vrednostno funkcijo stanja in stanja-akcije sedaj zapišemo

$$\hat{v}(s,w) \approx v_\pi(s),$$
$$\hat{q}(s,a,w) \approx q_\pi(s,a),$$

kjer w zdaj predstavlja vektor uteži, ki ga bomo dejansko spreminjali. Videli bomo, da so koristni predvsem funkcijski aproksimanti, ki so *odvedljivi* po parametrih w. Najpogosteje sta tako v uporabi **linearna aproksimacija** in **nevronske mreže**.

3.6.1. *Gradientni spust.* Najpogosteje uporabljena metoda »treniranja« oz. izboljšave aproksimantov je gradientni spust. Ideja je, da imamo neko funkcijo $J(w)$, ki je odvedljiva po vektorju parametrov w. Poiskati želimo njen (lokalni) minimum. To storimo tako, da izračunamo njen gradient $\nabla_w J(w)$ in nato premaknemo w v negativni smeri tega gradienta

$$\Delta w = -\alpha \nabla_w J(w),$$

kjer je α parameter hitrosti učenja (oz. velikost koraka). Čeprav ima ta parameter tu drugačen izvor kot pri algoritmih v tabelarični obliki, igra podobno vlogo, zato ima tudi enako oznako. Ker gradient pove, v katero smer funkcija narašča, bo premik v negativni smeri povročil, da bomo to vrednost zmanjšali. Parameter α je tu pomemben, saj prek njega lahko dosežemo, da se ne premaknemo predaleč in s tem ne zgrešimo iskanega minimuma. Ker spust začnemo z nekim naključnim začetnim približkom, lahko v splošnem dosežemo le lokalni minimum.

Za uporabo gradientnega spusta seveda potrebujemo neko kriterijsko funkcijo, ki jo želimo minimizirati. V našem primeru je smiselna izbira *srednja kvadratična napaka* med pravo vrednostjo stanja in ocenjeno vrednostjo, to je

$$J(w) = E_\pi[(v_\pi(s) - \hat{v}(s, w))^2],$$

kjer je π neka fiksna strategija, na podlagi katere ocenjujemo. Rešujemo torej problem napovedovanja. Uteži se potem prilagodijo v skladu z

$$\Delta w = \alpha E_\pi[(v_\pi(s) - \hat{v}(s, w))\nabla_w \hat{v}(s, w)],$$

kjer smo pri odvajanju pridobljen faktor 2 skrili v parameter α.

Seveda pa tu pridemo do dveh težav. Prva je uporaba pričakovane vrednosti, ki bi spet pomenila polno znanje o sistemu. To odpravimo z uporabo **stohastičnega gradientnega spusta**, kjer enostavno vzorčimo gradient

$$\Delta w = \alpha(v_\pi(s) - \hat{v}(s, w))\nabla_w \hat{v}(s, w).$$

V osnovnem primeru torej posodabljamo vrednosti glede na en sam vzorčni primer, kar se v resnici sklada s tabelaričnimi verzijami algoritmov. V povprečju vseeno dobimo enako posodobitev uteži.

Druga težava je zahteva po poznavanju $v_\pi(s)$, kar seveda pri problemu spodbujevalnega učenja ni možno. To težavo rešimo z dosedaj podano teorijo: $v_\pi(s)$ zamenjamo z eno izmed tarč, ki smo jih spoznali v poglavju 3. Najbolj pogosto je to tarča algoritma TD(λ), ki nam posodablja uteži v skladu z

$$\Delta w = \alpha(G_t^\lambda - \hat{v}(s, w))\nabla_w \hat{v}(s, w),$$

če uporabljamo algoritem s pogledom naprej oz.

$$\Delta w = \alpha \delta_t E_t$$

za

$$\begin{aligned} \delta_t &= R_{t+1} + \gamma \hat{v}(S_{t+1}, w) - \hat{v}(S_t, w), \\ E_t &= \gamma \lambda E_{t-1} + \nabla_w \hat{v}(S_t, w), \end{aligned}$$

če uporabljamo pogled nazaj. Do ostalih verzij algoritma pridemo na analogen način.

Problem upravljanja v primeru funkcijske aproksimacije rešujemo analogno osnovnemu primeru, kjer seveda uporabimo $\hat{q}(s, a, w)$ in ne $\hat{v}(s, w)$. Naletimo pa na nov

problem; konvergenca namreč ni več nujno zagotovljena, vendar je vseeno lahko dosežena v posebnih primerih.

3.6.2. *Linearna aproksimacija.* Najbolj enostaven način aproksimacije funkcij je seveda linearna aproksimacija. Ideja je, da stanju s določimo vektor

$$x(s) = \begin{bmatrix} x_1(s) \\ x_2(s) \\ \vdots \\ x_n(s) \end{bmatrix},$$

kjer je vsaka komponenta funkcija $x_i(s) : \mathcal{S} \to \mathbb{R}$.

Zgled 3.13 (Vektor $x(s)$ pri križcih in krožcih)**.** *Igralno ploščo križcev in krožcev lahko predstavimo kot vektor devetih števil, kjer* 1 *pomeni križec,* 0 *pomeni prazno polje in* -1 *pomeni krožec. Če želimo pridelati malo daljši vektor in s tem več uteži, se standardno uporabi binarni vektor stanja, ki je sestavljen le iz enic in ničel. Ideja je, da pripravimo vektor s* $27 = 3 \times 9$ *komponentami: prvih devet števil ima enice povsod, kjer so na plošči križci in drugod ničle, drugih devet ima enice na mestih krožcev in zadnjih devet na praznih mestih.*

Sedaj lahko predstavimo vrednost stanja kot linearno kombinacijo komponent vektorja stanja

$$\hat{v}(s, w) = x(s)^T w = \sum_{i=1}^{n} x_i(s) w_i.$$

V tej situaciji stohastični gradientni spust konvergira h globalnemu minimumu [11]. Dobimo tudi zelo enostavno pravilo posodabljanja uteži, saj velja $\nabla_w \hat{v}(s, w) = x(s)$. To zelo poenostavi računanje in razmislek o teoriji v ozadju. Zato je tudi linearna aproksimacija najbolj teoretično obdelana – konvergenčni dokazi za funkcijske aproksimacije so večinoma narejeni samo za linearni primer [12].

3.6.3. *Nevronske mreže.* Umetne nevronske mreže so v zadnjem času zelo popularne v nadzorovanem učenju zaradi uspehov pri problemih klasifikacije. Konceptualno so modelirane po možganih: vsebujejo skupke *nevronov*, ki so organizirani v *sloje*. Prvi sloj se imenuje vhodni in ima toliko nevronov, kot je velik naš vhodni podatek. Za problem križcev in krožcev smo v zgledu 3.13 videli, da je to potem 9 ali 27, če uporabljamo binarni vektor. Zadnji sloj je izhodni, vmes pa so *skriti* sloji, ki jih je lahko poljubno mnogo in so lahko poljubne velikosti. Vsi sloji so med sabo povezani, in sicer tako, da so povezani vsi nevroni iz enega sloja z vsemi nevroni iz sosednjega sloja. Lepota te strukture je, da je univerzalni aproksimator, če vsebuje skriti sloj in je dovolj velika. Torej lahko oceni poljubno zvezno funkcijo do katerekoli natančnosti [5].

Za velikost izhodnega sloja imamo v našem primeru dve možnosti. Lahko ima velikost 1, če želimo, da mreža vrača oceno stanja oz. para stanja in akcije, ki sta ji podana. Po drugi strani pa jo lahko naredimo tako veliko, kot imamo na voljo akcij in za vsako stanje ocenimo vse akcije hkrati.

Prehod med posameznimi sloji lahko seveda zapišemo malo bolj matematično. Naj a_n označuje *aktivacijo* n-tega sloja. To je torej vektor, ki predstavlja, kakšne vrednosti je dosegel vhodni podatek, ko je prišel do n-tega sloja. Vsak nevron ima svoj vektor uteži. Vse vektorje nevronov n-tega sloja zložimo v matriko W_n velikosti $(velikost\ sloja\ n) \times (velikost\ sloja\ n+1)$. Vsak nevron ima načeloma tudi svojo

pristranskost (angl. *bias*), ki jo za posamezen sloj označimo z vektorjem b_n. Sedaj lahko končno napišemo, kako se spremeni vrednost pri prehodu med slojema:

$$a_{n+1} = f(W_n a_n + b_n).$$

Pri tem je f *nelinearna monotona* funkcija, ki ji pravimo **aktivacijska funkcija**. V model vnese nelinearnost, kar naredi nevronske mreže tako uporabne in univerzalne.

Zgled 3.14 (Zgledi aktivacijskih funkcij)**.** *Aktivacijske funkcije, ki so uporabljene najpogosteje, so:*

- *Sigmoidna funkcija:* $f(x) = \frac{1}{1+e^{-x}}$,
- *Hiperbolični tangens:* $f(x) = \frac{1-e^{-2x}}{1+e^{-2x}}$,
- *ReLU:* $f(x) = \max(0, x)$.

Te funkcije so nato v nevronski mreži uporabljene po komponentah.

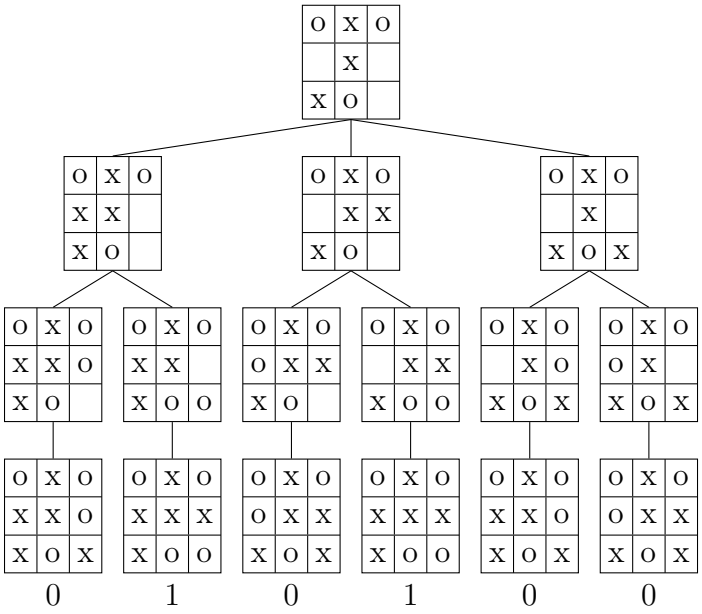
Kljub velikim uspehom v zadnjem času, imajo nevronske mreže vseeno mnogo težav. Veliko parametrov je potrebno nastaviti ročno, še več pa jih je potrebno prilagoditi prek učenja. Naletimo lahko na prekomerno prilagajanje podatkom ali pa dobimo konvergenco k lokalnemu minimumu. Poseben problem pri spodbujevalnem učenju pa je, da z uporabo nevronskih mrež kot aproksimanta v primeru algoritmov TD(0) in TD(λ) konvergenca ni zagotovljena [11].

3.6.4. *Serijsko učenje (angl. Batch reinforcement learning).* Ena izmed nadgradenj, ki v praksi izboljša konvergenco algoritmov, je serijsko učenje. Veliko uspeha je dosegla predvsem metoda ponovitve izkušenj (angl. *experience replay*), kar v osnovi pomeni, da v tabelo hranimo zadnjih nekaj opažanj kombinacij stanj, akcij in nagrad in se učimo s pomočjo naključne izbire iz te tabele namesto iz iger direktno. S tem se znebimo korelacije med podatki [6]. Tovrstne nadgradnje presegajo obseg tega dela.

4. Namizne igre

Namizne igre (npr. šah, go, križci in krožci, ...) so proces, ki se zelo lepo poda modeliranju z Markovskim procesom odločanja. Množico stanj $\mathcal{S}$ enostavno zapolnimo z vsemi možnimi konfiguracijami igralne plošče. Množico akcij $\mathcal{A}$ sestavljajo vse legalne poteze, ki jih definirajo pravila igre in trenutno stanje. Prehodno matriko definirajo akcije našega nasprotnika oz. okolja, če je igra stohastične narave (npr. Backgammon - met kocke). Ker ne poznamo nasprotnika, so namizne igre okolje, kjer je prehodna matrika neznana in zato učenje prek dinamičnega programiranja ne pride v poštev. Prav tako samo delno poznamo nagradno funkcijo, saj je tudi ta odvisna od dinamike okolja.

4.1. **Formalna definicija namizne igre.** Namizne igre, s katerimi se ukvarjamo v tem diplomskem seminarju, sodijo v skupino *kombinatornih iger*. To so igre za dva igralca s popolno informacijo (noben aspekt igre igralcem ni skrit), kjer igralca izmenično igrata poteze. Tovrstnim igram se odlično poda reprezentacija z drevesom, kjer je vsako vozlišče možno stanje (možna stanja določajo pravila igre), povezave pa predstavljajo poteze igralcev. Primer takega drevesa vidimo na sliki 2.



Slika 2. Del drevesa igre križci in krožci.

4.2. **Nagrajevanje.** Pri namiznih igrah je najpogosteje uporabljena zelo enostavna struktura nagrad. Zmagi in porazu sta dodeljeni vrednosti 1 oz. -1, za izenačenje se vzame vrednost 0. Vsem ostalim ne-končnim stanjem se prav tako dodeli vrednost 0 in tako se doseže, da imamo opravka z igro z vsoto nič tudi znotraj MDP-ja. Občasne spremembe te strukture nagrad so možne, najpogosteje se dodeli neko negativno nagrado vsem stanjem (npr. $-0,1$). Konceptualno naj bi to pomenilo pritisk na agenta, da naj čim prej pride do konca igre, tj. naj čim prej zmaga. Možna je tudi neničelna nagrada za izenačenje, njen predznak pa je odvisen od tega, ali je izenačenje želen izid.

Posebnost pri namiznih igrah je tudi v diskontnem faktorju. Čeprav so vseeno zaželene nagrade, ki pridejo prej, torej krajše igre, se zaradi pojavitve neničelnih nagrad samo ob koncu igre pogosto uporablja diskontni faktor $\gamma = 1$.

Namizne igre v kontekstu te diplomske naloge so igre z vsoto nič. To pomeni, da se nagrade igralcev seštejejo v 0. Za namene spodbujevalnega učenja lahko agentove nagrade odstopijo od te konvencije (npr. z že omenjeno malo negativno nagrado ob vsakem koraku), a v osnovi imamo opravka s tovrstnimi igrami.

4.3. **Optimalna strategija.** Če imamo opravka z okoljem za enega agenta, je ideja optimalne strategije dokaj enostavna – to je strategija, ki agentu dolgoročno prinese najvišjo nagrado. Namizne igre pa imajo drugačno zasnovo; so igre za dva igralca oz. agenta. Sedaj je optimalna strategija enega odvisna od optimalne strategije drugega. Tu nam na vprašanje optimalne strategije ponovno pomaga odgovoriti teorija iger.

Vrednostna funkcija v modelu z dvema agentoma ocenjuje njuno skupno strategijo, označeno s $\pi = (\pi_1, \pi_2)$. Komponenti sta še vedno definirani na enak način kot v osnovnem modelu. Razlika je v optimalni vrednostni funkciji, ki ji tu pravimo **minimax vrednostna funkcija**. Ta funkcija maksimizira nagrado za prvega igralca

in minimizira nagrado za drugega in je definirana kot

$$v_*(s) = \max_{\pi_1} \min_{\pi_2} v_\pi(s),$$

kjer je $v_\pi(s) = \mathrm{E}[G_t \mid S_t = s]$. v_* opisuje vrednost stanja z vidika prvega igralca. Ker pa imajo preučevane igre vsoto nič, dobimo vrednost z vidika drugega igralca tako, da pomnožimo $v_*(s)$ z -1. **Minimax strategija** je potem skupna strategija, ki doseže vrednosti minimax vrednostne funkcije. Kot posledica znanega rezultata iz teorije iger, izreka o minimaxu, je minimax vrednostna funkcija ena sama, minimax strategija, ki jo določa, pa je Nashevo ravnotežje. Tu je pomembno dejstvo, da govorimo o igrah z vsoto nič, ki imajo popolno informacijo.

4.4. **Parcialni model – »po-stanja«.** V veliki večini primerov je pri spodbujevalnem učenju uporabna vrednostna funkcija akcij q, kar je zaradi neznane dinamike okolja in nepredvidljivosti njegovih odgovorov smiselno. Vrednostna funkcija stanj v nam ne pomaga pri izboljšavi strategije, saj nam ena vrednost stanja ne pove nič o tem, katera akcija je dobra, govori samo o tem, kakšno je trenutno stanje. Za namene učenja pa je pomembno vedeti, kako dobre so posamezne akcije v stanju, ali pa še boljše, kako dobro je najboljše dosegljivo stanje.

Izkaže se, da so kombinatorne igre primer okolja, kjer točno vemo, v katero stanje nas bo pripeljala katera akcija – imamo nek parcialni model. Zato se tu ocenjevanja vrednosti lotimo drugače. Ocenjujemo namreč vrednosti »po-stanj« (angl. *afterstates*). To so stanja, v katera nas pripelje določena akcija. S pomočjo tega dejstva lahko za učenje uporabimo funkcijo v. Ker je stanj manj kot parov stanj in akcij, ta zamenjava zmanjša velikost tabele, v katero funkcija shranjuje vrednosti in tako pospeši učenje.

Zgled 4.1. *Na začetku igre križcev in krožcev je plošča popolnoma prazna. Ko se prvi igralec odloča, kam bo postavil svoj križec, nam vrednost prazne plošče ne pomaga nič, vrednosti vseh dosegljivih stanj (po-stanj) in vrednosti vseh akcij, ki jih ima na voljo, pa nosijo enako informacijo.*

4.5. **Pridobivanje iger.** Še ena posebnost pri spodbujevalnem učenju je generiranje iger oz. epizod. Imamo namreč dva zelo različna pristopa.

4.5.1. *Samoigra.* Če se vedemo, da je namizna igra okolje za dva agenta, je najbolj naraven in enostaven način za pridobivanje epizod samoigra. To pomeni, da dva identična agenta igrata drug proti drugemu in se soprilagajata. Tak način generiranja iger se tudi najlepše prilega teoriji iger, ki tiči v ozadju. Slabost pa je, da agenta začneta brez znanja in potrebujeta veliko količino iger, da prenehata z igro, ki je skoraj naključna.

4.5.2. *Okolje z enim agentom.* Alternativa je, da simuliramo okolje s samo enim agentom. To ponavadi naredimo na enega od treh načinov, ki jih opisuje Ghory [3]:

- **Igre iz podatkovnih baz.** Za nekatere popularne igre obstajajo podatkovne baze, kjer so shranjene stare igre. Te so lahko profesionalne (npr. iz kakšnih svetovnih prvenstev) ali pa samo naključne igre med navadnimi ljudmi. Vsekakor je to odlična alternativa samoigri, saj so igre iz podatkovnih baz pogosto zelo kvalitetne, kar omogoča hitrejše učenje agenta. Prav tako je potrebno hraniti in posodabljati vrednostno funkcijo samo za enega agenta. kar malo izboljša prostorsko in računsko zahtevnost.

- **Naključen nasprotnik.** Če podatkovno bazo zamenjamo z generatorjem naključnih potez, dosežemo podobno zmanjšanje računske zahtevnosti, a povzročimo nekaj novih težav. Izkaže se, da v mnogo igrah obstajajo strategije, ki zelo enostavno premagajo naključnega igralca, a izgubijo proti praktično kateremukoli človeku. Prav tako je naključna igra ponavadi zelo različna od stila igranja človeka. Naključni igralec ima lahko težave tudi pri uporabi nekaterih pravil igre in posledično privede do nepotrebnega podaljšanja epizod.
- **Fiksen nasprotnik.** Najenostavnejša alternativa samoigri je uporaba fiksnega nasprotnika. Ta je ponavadi računalnik, v redkih primerih pa tudi človek. Tu se pojavi problem prevelikega prilagajanja strategije temu nasprotniku. Pogosto se zgodi, da se agent nauči strategije, ki premaga fiksnega nasprotnika, a izgubi proti drugim nasprotnikom.

4.6. **Primer algoritma za namizne igre.** Če nasprotnik ni kopija agenta, je algoritem zelo podoben algoritmu 2, saj je simulirano okolje z enim agentom. Do razlik pride, če je metoda učenja samoigra. V algoritmu 3 je prikazano, kako izgleda algoritem TD(0) z učenjem ob koncu posamezne epizode/igre (»offline«).

Algorithm 3 TD(0) s samoigro – uporaba po-stanj

Podatki: parameter hitrosti učenja α, število epizod $stEpizod$, diskontni faktor γ, parameter požrešnosti ϵ

Poljubno nastavimo vrednosti $V(s)$ za vsak $s \in \mathcal{S}$ (npr. $V(s) = 0$ za vsak $s \in \mathcal{S}$)

for $k = 1, 2, \ldots, stEpizod$ **do**
  Generiraj epizodo prek funkcije po-stanj V ϵ-požrešno
  $stanja1 \leftarrow$ seznam vseh opaženih stanj po potezi prvega igrlaca
  $stanja2 \leftarrow$ seznam vseh opaženih stanj po potezi drugega igralca
  $nagrade1 \leftarrow$ seznam vseh opaženih nagrad po potezi prvega igralca
  $nagrade2 \leftarrow$ seznam vseh opaženih nagrad po potezi drugega igralca

  for $t = 1, 2, \ldots, length(stanja1) + length(stanja2)$ **do**
    if jeLih(t) **then**
      $s1 = stanja1[t]$
      $s1' = stanja1[t+1]$
      $V(s1) \leftarrow V(s1) + \alpha(nagrade1[s1'] + \gamma V(s1') - V(s1))$
    else
      $s2 = stanja2[t]$
      $s2' = stanja2[t+1]$
      $V(s2) \leftarrow V(s2) + \alpha(nagrade2[s2'] + \gamma V(s2') - V(s2))$
    end if
  end for
end for

Po-stanja spremenijo algoritem tako, da na prvi pogled izgleda, da se agent uči za stanja, v katerih akcijo izbira nasprotnik namesto on sam. Ta sprememba nam dovoli, da se izognemo računanju vrednostne funkcije akcij.

4.7. **Nekatere nadgradnje za namizne igre.** Za reševanje namiznih iger je najbolj znan **minimax algoritem**. V osnovi je to iskanje po drevesu igre. Zaradi eksponentne rasti drevesa se algoritem naredi uporaben tako, da se išče do fiksne globine, nato pa se vrednosti stanj oceni z neko funkcijo. Izkaže se, da lahko tako iskanje združimo s spodbujevalnim učenjem in to funkcijo spremenimo v poznano v. To nekoliko izboljša učinkovitost algoritmov, a presega obseg te naloge. Metode natančneje opisujeta Silver [6] in Ghory [3].

5. Aplikacija v praksi

Spoznano teorijo smo aplicirali na praktičen primer namizne igre. V tem odseku bomo predstavili nekaj rezultatov, do katerih smo prišli. Vsa koda in dodatni rezultati so prosto dostopni na https://github.com/timkalan/diploma.

5.1. **m,n,k-igra.** m,n,k-igra je abstraktna namizna igra, kjer dva igralca drug za drugim na ploščo z m vrsticami in n stolpci polagata kamenček svoje barve (oz. »križce« in »krožce«). Njun cilj je, da v vrsto postavita k kamenčkov svoje barve. Vrsta kamenčkov je lahko v vrstici, stolpcu ali na diagonali. Kdor prvi uspe, zmaga.

Zaradi naravne simetrije igre ima praviloma prvi igralec prednost, saj naredi prvo potezo. Ker nobena poteza ni škodljiva za igralca, ki jo je igral, po argumentu kraje strategije ne obstaja strategija, ki bi zagotovila zmago drugemu igralcu [4]. To pomeni, da se m,n,k-igre ob predpostavki popolne igre s strani obeh igralcev končajo bodisi z zmago prvega igralca, bodisi z izenačenjem (odvisno od m, n in k).

Možna in enostavna prilagoditev igre je dodatek *gravitacije*. To pomeni, da se v vsakem stolpcu kamenček lahko postavi le v najnižjo vrstico.

Zgled 5.1. *Igra križci in krožci so* 3,3,3*-igra. Popularna japonska igra gomoku je* 15,15,5*-igra. Ob dodatku gravitacije je igra štiri v vrsto (angl. Connect four)* 6,7,4*-igra.*

5.1.1. *Kompleksnost m,n,k-igre.* Merjenje kompleksnosti iger je globok problem. Mi se ga bomo samo dotaknili, saj nam lahko da idejo o velikostnih razredih problemov, ki jih rešujemo. Kompleksnost bomo merili izključno v smislu števila stanj, to pa seveda ni edini način. Točno število stanj za m,n,k-igro je v splošnem težko izračunljivo, lahko ga pa enostavno ocenimo kot $zgornja\ meja = 3^{mn}$. Logika v ozadju je, da ima vsako polje lahko eno izmed treh vrednosti: križec, krožec ali pa je prazno. Ta metoda upošteva mnogo nedovoljenih stanj (npr. celotna plošča prekrita s križci), služi nam samo za orientacijo.

5.2. **Metoda.**

5.2.1. *Algoritmi.* Ogledali smo si tabelarnega agenta in agenta, kjer je vrednostna funkcija aproksimirana z nevronsko mrežo. Za samo učenje smo uporabili algoritme TD(0) in TD(λ) in MC. Pod drobnogled smo vzeli 3,3,3-igro in 5,5,4-igro.

Tabelarni agent vrednostno funkcijo hrani v obliki slovarja, kjer so ključi vsa obiskana stanja. Parameter hitrosti učenja α in parameter raziskovanja ϵ sta bila prek poskušanja nastavljena na učinkoviti vrednosti – to sta $\alpha = 0,2$ in $\epsilon = 0,3$. Parameter λ pri algoritmu TD(λ) smo nastavili na $0,9$.

Pri agentu s funkcijsko aproksimacijo sta se učinkovita parametra nekoliko razlikovala, $\alpha = 1$ in $\epsilon = 0,05$. Spet smo ju pridobili s pomočjo poskušanja. Za uporabo nevronskih mrež pa moramo določiti nekaj stvari, ki niso direktno povezane z uporabljenimi algoritmi. Izbira velikosti in globine nevronske mreže je pogosto arbitrarna

izbira, mi smo izbrali mrežo z dvema skritima slojema. Velikost vhodnega sloja je določena z velikostjo igre. Ploščo smo predstavili z binarnim vektorjem stanja in na koncu dodali še nevron pristranskosti (angl. *bias neuron*). Tako je formula za velikost $3 \times m \times n + 1$. Velikost obeh skritih slojev je dvakratnik velikosti vhodnega sloja. Izhodni sloj pa ima enostavno velikost 1, saj nas zanima samo ocena stanja, ki ga podamo mreži.

Pomembna izbira je tudi aktivacijska funkcija. S pomočjo eksperimentiranja smo ugotovili, da nam najbolje služi funkcija ReLU oz. $f(x) = \max(0, x)$.

Poslužili smo se tudi standardnega diskontnega faktorja za namizne igre, to je $\gamma = 1$. Struktura nagrad je standardna za namizne igre. To pomeni 1 za zmago, -1 za poraz in 0 sicer.

5.2.2. *Pripomočki.* Uporabili smo programski jezik *Python*. Nekatere numerične operacije so pospešene s pomočjo knjižnice za numerično računanje *NumPy*. Nevronskih mrež nismo implementirali sami, pač pa smo uporabili knjižnico *PyTorch*.

5.2.3. *Postopek.* Za kombinacije iger in algoritmov smo izvedli enakovreden poskus. Uporabili smo metodo samoigre, s pomočjo katere so agenti igrali vnaprej določeno število iger. Na vsake 500 iger smo učenje ustavili, pridobljeno strategijo agenta pa uporabili za 1000 iger proti naključnemu igralcu. To smo ponovili za agenta v vlogi začetnega in drugega igralca. Zmage, izenačenja in poraze smo potem narisali na graf. Oznake grafov so vedno napisane z vidika agenta. Npr. »zmage« pomenijo zmage agenta (ne glede na to, ali je agent prvi ali drugi igralec). Naslov grafa pa je napisan tako, da je na prvem mestu omenjen igralec, ki je začel in na drugem nasprotnik (ponovno ne glede na to, kateri igralec je agent in kateri je naključni).

5.3. **3,3,3-igra.** Igra križci in krožci je znana otroška igra. Okvirna zgornja meja za količino stanj je 19683. Splošno je znano, da se ob predpostavki popolne igre s strani obeh igralcev konča z izenačenjem. Ker so v našem primeru testi izvedeni proti naključnemu igralcu, bo ob predpostavki uspešnega učenja večino iger zmagal agent. Za nas pomembnejši pokazatelj kvalitete strategije pa je količina porazov. Teh pri uporabi popolne strategije ni. Stanje agentov pred učenjem je prikazano v tabeli 1. Agenti so poimenovanu po algoritmu ki je v ozadju. Če je dodana funkcijska aproksimacija, se imenu doda »NM«, kar pomeni *nevronska mreža*. Številka pri imenu pa nam pove, ali je agent prvi ali drugi igralec.

IGRALEC	ZMAGE [%]	IZENAČENJA [%]	PORAZI [%]
TD(0) 1.	78,1	4,22	17,7
TD(λ) 1.	78,1	4,16	17,7
MC 1.	78,0	4,17	17,9
TD(0)-NM 1.	43,4	20,9	35,7
TD(0) 2.	43,9	3,79	52,3
TD(λ) 2.	44,2	3,84	52,0
MC 2.	44,3	3,82	51,9
TD(0)-NM 2.	16,3	25,5	58,2

TABELA 1. Rezultati igranja proti naključnemu igralcu pred učenjem agenta.

Stanje v tabeli je praktično igra med dvema naključnima igralcema. Tu lahko empirično vidimo, da ima prvi igralec res prednost, saj zmaga večino iger. Izjema je agent z nevronsko mrežo, kjer naključna nastavitev njenih uteži privede do nekoliko drugačne igre.

V tabeli 2 lahko vidimo še koliko epizod učenja je potreboval posamezni agent, da med igranjem tisočih iger proti naključnemu igralcu nikoli ni izgubil. To seveda ni popoln pokazatelj kvalitete igre, saj je lahko to samo naključna sreča, a vseeno nam da okvirno idejo in pa primerjavo med algoritmi.

IGRALEC	KOLIČINA IGER
TD(0) 1.	1500
TD(λ) 1.	1500
MC 1.	3000
TD(0)-NM 1.	8500

IGRALEC	KOLIČINA IGER
TD(0) 2.	17500
TD(λ) 2.	21500
MC 2.	23000
TD(0)-NM 2.	133500

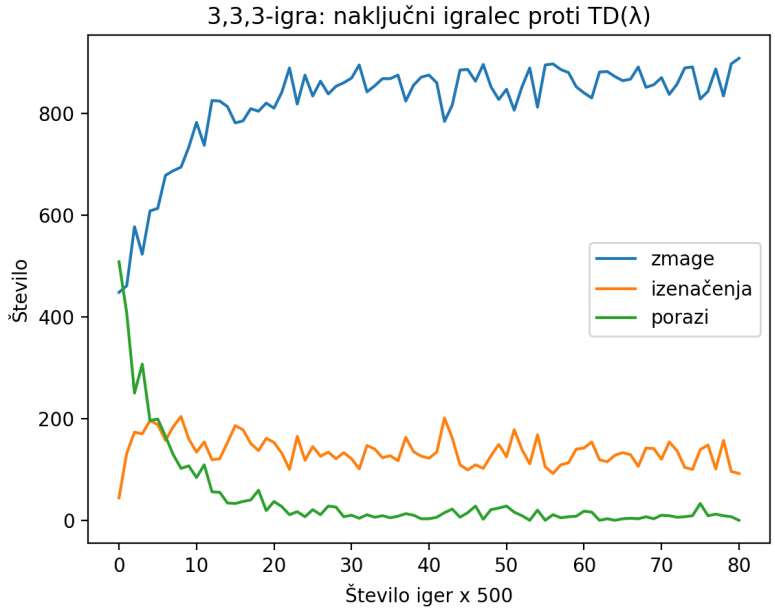
Tabela 2. Količina potrebnih iger, da algoritem prvič doseže 0 porazov v 1000 ponovitvah igre.

Tabela nam pokaže, da vsaj za ta problem ni bistvene razlike med obravnavanimi tabelarnimi algoritmi. To dejstvo je bilo tudi vodilo, da smo za nevronske mreže prilagodili samo algoritem TD(0). Ta pa je za doseg optimalne igre potreboval občutno več časa kot tabelarna verzija.

Za večjo nazornost grafov smo se odločili, da bomo agenta pri tej igri vedno postavili v vlogo drugega igralca, to pa pomeni, da tudi dobi težjo nalogo – v naših eksperimentih se je izkazalo, da za učenje dobre strategije v tej vlogi agent potrebuje občutno več iger, kot če je v vlogi prvega igralca. Grafi zamenjanih vlog izgledajo podobno, le da je porazov že pred učenjem manj.

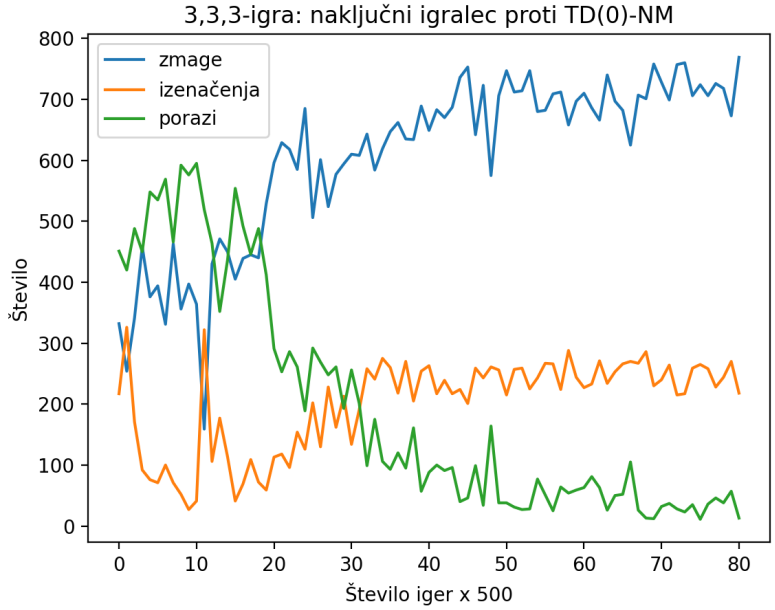
5.3.1. *Tabelarni agent.* Najprej smo se problema lotili na bolj osnoven način – s pomočjo vrednostnih funkcij v obliki tabel. Uporabili smo prej omenjeno vrednostno funkcijo po-stanj in jo predstavili kot slovar v Pythonu, ki se je polnil z vsakim novim odkritim stanjem.

Na grafu 3 lahko vidimo primer procesa učenja enega izmed naših agentov, to je algoritem TD(λ) v vlogi drugega igralca. Ključen element grafa je število porazov. Želimo, da pada proti 0 in to tukaj tudi dosežemo. Ker pa smo se odločili za uporabo konstantnega ϵ in α, pa tudi ob veliki količini iger občasno dobimo neoptimalno igro – pri posodabljanju vrednosti je narejen prevelik korak oz. premik, to pa privede agenta do tega, da zgreši optimum.



Slika 3. Proces učenja TD(λ) v vlogi drugega igralca tekom 40000 iger.

5.3.2. *Agent s funkcijsko aproksimacijo – nevronske mreže.* Zaradi relativno majhnega števila stanj v tej igri je tabelarni agent tu hitreje uspešen, saj hitro odkrije večino stanj. Agent z nevronsko mrežo pa ne more beležiti vrednosti dejanskih stanj, kar pomeni, da dobro strategijo lahko odkrije šele, ko se večina parametrov nevronske mreže nastavi na smiselne vrednosti. To pa zaradi velike količine parametrov traja dlje, kot traja samo učenje tabelarnega agenta. Vseeno pa na koncu tudi ta agent pride do strategije, ki proti naključnemu igralcu ne izgubi nobene igre.

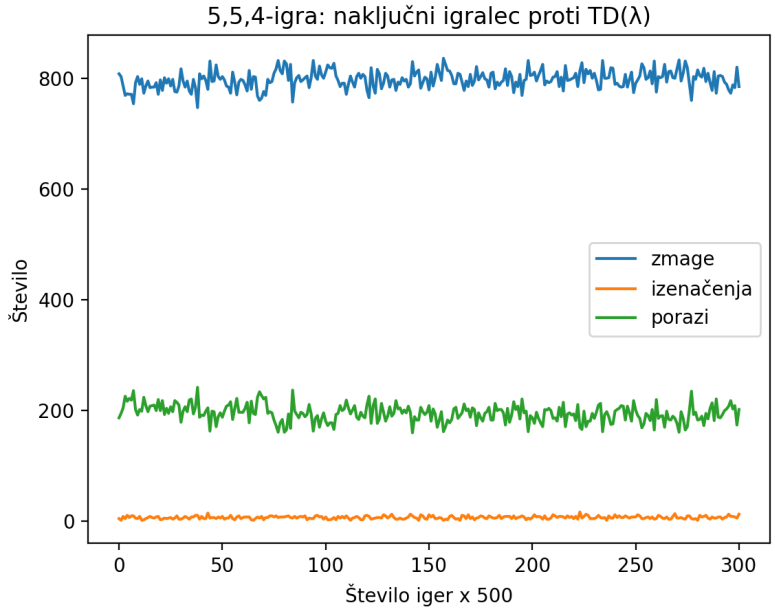


Slika 4. Proces učenja TD(0) z nevronsko mrežo v vlogi drugega igralca tekom 40000 iger.

5.4. **5,5,4-igra.** Če povečamo velikost igre, tabelarne metode hitro odpovedo (kot bomo videli v nadaljevanju). V tem primeru lahko grobo ocenimo zgornjo mejo za količino stanj kot $3^{(5\times5)} \approx 8 \times 10^{11}$, kar je 43 milijonkrat več, kot je zgornja meja za križce in krožce. Tabelarni agent bi moral igrati zelo veliko iger, da bi uspel raziskati tako velik prostor stanj. V tem primeru pride zelo prav sposobnost generalizacije agenta z nevronsko mrežo. Generalizacija za nas pomeni, da se tudi neobiskanim stanjem določi neko vrednost, ki je smiselna, če je problem pravilno zastavljen.

5,5,4-igra se pri predpostavki popolne igre obeh igralcev konča z izenačenjem. Na grafih, ki prikazujejo igre proti naključnemu igralcu, zato pričakujemo, da bi naučeni agent zmagal večino iger. Ponovno je naš dejanski cilj doseči čim manj porazov. Glede na relativno velikost igre pričakujemo, da bo izenačenj manj, kot pri križcih in krožcih. Tam je možnih stanj tako malo, da se pogosto zgodi, da naključni igralec ponesreči prepreči zmago agenta. V 5,5,4-igri je stanj veliko več in posledično manj verjetno, da opazimo tako vedenje naključnega igralca.

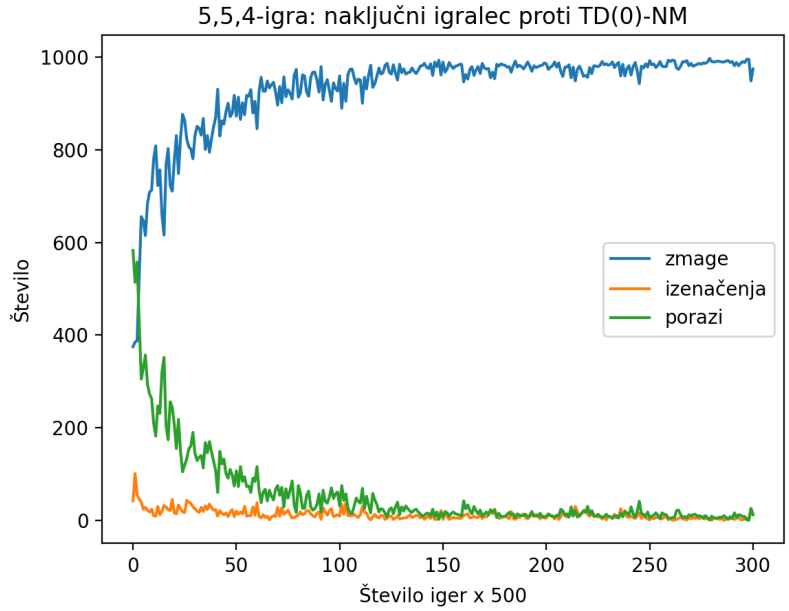
5.4.1. *Tabelarni agent.* Pri tako velikem prostoru stanj tabelarni agent popolnoma odpove. To je razvidno tudi v grafu 5, kjer vidimo, da se tudi po 150000 igrah agentova strategija ne izboljša.



Slika 5. Neuspeh učenja tabelarnega TD(λ) agenta.

Na grafu 5 opazimo tudi, da je bila naša hipoteza o manjši količini izenačenih iger pravilna. Do izenečenja ni prišlo praktično nikoli.

5.4.2. *Agent s funkcijsko aproksimacijo – nevronske mreže.* Funkcijska aproksimacija se v tem primeru izkaže za bolj uspešno. Na grafu 6 vidimo, da je ta agent neprimerljivo uspešnejši od tabelarnega. Dobljeni rezultat se sklada z našimi pričakovanji. Dobimo strategijo, ki uspešno premaga vsaj naključnega igralca.



Slika 6. Proces učenja TD(0) z nevronsko mrežo v vlogi drugega igralca tekom 150000 iger.

5.5. **Razprava.** Zgornja primera iger dobro prikažeta razlike med algoritmi, ki uporabljajo funkcijsko aproksimacijo in algoritmi, ki shranjujejo vrednosti v tabelah. Pri manjših in manj zahtevnih problemih se izkaže, da je uporaba nevronskih mrež nepotrebna in nam celo škoduje. Za zahtevnejše probleme pa aproksimacija postane nepogrešljiva izboljšava, ki nam dovoli, da se problem sploh začne reševati.

Pri popularnih resnih namiznih igrah je navada, da se dobljeni algoritem primerja s človeškimi igralci. To se naredi s pomočjo t. i. Elo sistema točkovanja. Tovrstno točkovanje za naša primera ne obstaja, zato je primerjava s človeškimi igralci nemogoča.

Algoritem smo poskusili aplicirati tudi na še večjem problemu, to je igri štiri v vrsto. Naše metode so se izkazale za neuspešne, nevronska mreža je divergirala po določeni količini iger pri vseh poskušenih vrednostih parametrov. Težave se lahko vseeno skrivajo v parametrih, velikosti mreže ali pa so te metode brez izboljšav enostavno nezadostne za reševanje tako velikih problemov. Odgovor je verjetno kombinacija vseh treh razlogov, vemo pa vsaj, da so bile metode spodbujevalnega učenja z nekaterimi prilagoditvami odgovorne za računalniški program, ki je v igri go prvi premagal svetovnega prvaka [7].

Slovar strokovnih izrazov

reinforcement learning spodbujevalno učenje
state-value function vrednostna funkcija stanja
action-value function vrednostna funkcija akcije
Markov decision process (MDP) Markovski proces odločanja
temporal-difference learning učenje s časovno razliko
eligibility trace sled upravičenosti
(artificial) neural network (umetna) nevronska mreža

Literatura

[1] R. E. Bellman. *Dynamic Programming*. Princeton University Press, Princeton, 1957.

[2] R. E. Bellman. A markov decision process. *Journal of Mathematical Mechanics*, 6, 1957.

[3] I. Ghory. Reinforcement learning in board games, 2004.

[4] A. W. Hales in R. I. Jewett. Regularity and positional games. *Transactions of the American Mathematical Society*, 106, 1963.

[5] K. Hornik, M. Stinchcombe in H. White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2, 1989.

[6] D. Silver. Introduction to reinforcement learning. https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver, 2015.

[7] D. Silver idr. Mastering the game of go with deep neural networks and tree search. *Nature*, 529, 2016.

[8] S. Singh, T. Jaakkola, M. L. Littmanm in C. Szepesvari. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 39, 2000.

[9] S. Singh in R. S. Sutton. Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22, 1996.

[10] R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3, 1988.

[11] R. S. Sutton in A. G. Barto. *Reinforcement Learning: An introduction*. The MIT Press, Cambridge, Massachusetts, 2 edition, 2015.

[12] C. Szepesvari. *Algorithms for Reinforcement Learning*. Morgan & Claypool Publishers, Alberta, Canada, 2009.