Univerza v Ljubljani
Fakulteta za računalništvo in informatiko
Fakulteta za matematiko in fiziko

Alen Bizjak

Igranje igre "Kamen, papir, škarje, kuščar, Spock" z metodami umetne inteligence

diplomsko delo

interdisciplinarni univerzitetni
študijski program prve stopnje
računalništvo in matematika

Mentor: prof. dr. Zoran Bosnić

Ljubljana, 2020

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Kandidat: Alen Bizjak

Naslov: Igranje igre Kamen, papir, škarje, kuščar, Spock z metodami umetne inteligence

Opis:

Kandidat naj v diplomskem delu obravnava igranje igre kamen, papir, škarje, kuščar, Spock z metodami umetne inteligence. Po pregledu sorodnih del naj implementira, razvije lastne in primerja različne algoritme za igranje te igre. Njihovo uspešnost naj ovrednoti v tekmovanjih algoritmov medseboj in proti različnim statičnim (umetnim) zaporedjem nasprotnikovih potez. V delu naj kandidat predstavi uspešnosti posameznih algoritmov in ugotovitve glede njihove zgornje meje uspešnosti.

Kazalo

Seznam uporabljenih kratic

kratica	angleško	slovensko
KPŠ	rock, paper, scissors	kamen, papir, škarje
KPŠKS	rock, paper, scissors, lizard, Spock	kamen, papir, škarje, kuščar, Spock
VP	always counter	vedno premagaj
VPS	always counter self	vedno premagaj sebe
VO	always loop	vedno obračaj
VK	always rock	vedno kamen
MUPN	history string matching	metoda ujemanja preteklih nizov
MV	markov chains	markovske verige
SUPP	reinforcement learning based on moves	spodbujevano učenje na podlagi potez
MKSU	meta-classificator with reinforcement learning	meta-klasifikator s spodbujevanim učenjem
MKIA	meta-classificator with chosen algorithms	meta-klasifikator z izbranimi algoritmi

Povzetek

Naslov: Igranje igre "Kamen, papir, škarje, kuščar, Spock" z metodami umetne inteligence

Avtor: Alen Bizjak

Igranje igre "Kamen, papir, škarje, kuščar, Spock" ni zahtevno opravilo, saj je igra preprosta in na prvi pogled podvržena naključju. Bolj zahtevno pa je zmagati oziroma zmagovati na dolgi rok. Z metodami umetne inteligence smo v diplomskem delu na podlagi zgodovine odigranih potez v dani igri poskušali odkriti vzorce v nasprotnikovi strategiji in si zagotoviti pozitivno razmerje zmag in porazov. Pri tem smo se osredotočili izrecno na zgodovino odigranih potez. Ostalih možnih vhodov, kot na primer opazovanje gibanja nasprotnikove roke pri formaciji poteze, nismo uporabili.

Za ta namen smo razvili naslednje algoritme: metoda ujemanja preteklih nizov, markovske verige, spodbujevano učenje na podlagi potez in meta-klasifikator na podlagi spodbujevanega učenja. Razvite algoritme smo testirali preko različnih testnih scenarijev, ki so vključevali strojno učenje s pomočjo preprostih pomožnih algoritmov in vnaprej generirane nize potez.

Po izvedbi vseh eksperimentov smo algoritme med seboj primerjali in analizirali njihovo uspešnost. Rezultati so pokazali, da sta se v večini primerov najbolje izkazala algoritma markovske verige in meta-klasifikator na podlagi spodbujevanega učenja.

Ključne besede: umetna inteligenca.

Abstract

Title: Playing game "Rock, paper, scissors, lizard, Spock" using methods of artificial intelligence

Author: Alen Bizjak

Playing the game "Rock, Paper, Scissors, Lizard, Spock" is not difficult since the game is both simple and, at first glance, subject to chance. It is more challenging to develop a winning strategy that will guarantee a win or many wins in the long run. In this thesis, we used artificial intelligence to try to discover a pattern in the moves of the opponent, focusing specifically on the history of moves in a game to secure a positive win/loss ratio. We did not include other factors, such as observing the movement of the opponent's hand as they choose which move to use.

For this purpose we developed the following algorithms: history string matching, markov chains, reinforcement learning based on moves and meta-classificator with reinforcement learning. We tested our developed algorithms through various test scenarios, which included machine learning with the help of simple auxiliary algorithms and move sequences that were generated in advance.

After conducting all experiments, we compared the algorithms and analyzed their performance. The results have shown that in most test scenarios the best performing algorithms were markov chains and meta-classificator with reinforcement learning.

Keywords: artificial intelligence.

Poglavje 1

Uvod

Igranje povsem naključne igre s pomočjo umetne inteligence ne bi bilo zanimivo. Vsem znana igra Kamen, papir, škarje je na prvi pogled podvržena popolnemu naključju, vendar ni tako. Človeški igralec zavestno ali podzavestno tvori vzorce oziroma strategijo v svoji igri, ki jih z metodami umetne inteligence lahko poskušamo predvideti.

Namesto osnovne variante smo v sklopu diplomske naloge obravnavali njeno razširitev: Kamen, papir, škarje, kuščar, Spock. Tu osnovnim trem potezam (kamen, papir in škarje) dodamo še dve: kuščar in Spock. Vsaka poteza premaga dve potezi (kamen premaga škarje in kuščarja, kuščar premaga papir in Spocka, Spock premaga škarje in kamen, škarje premagajo papir in kuščarja, papir premaga kamen in Spocka) in ostaja izenačena proti sama sebi - simetričnost originalne igre je tako ohranjena.

Poleg tega smo namesto proti človeškemu igralcu igrali proti različnim računalniško generiranim nizom ter proti različnim algoritmom. Tak pristop je potreben, saj implementirani algoritmi za uspešno delovanje potrebujejo dovolj veliko učno množico, ki jo igranje proti človeškemu igralcu ne bi zagotovilo.

Implementirali smo devet algoritmov (v nadaljevanju tudi agentov), ki smo jih razdelili v dve skupini. Prvo skupino sestavljajo metoda ujemanja preteklih nizov, markovske verige, spodbujevano učenje na podlagi potez in

meta-klasifikator na podlagi spodbujevanega učenja. Druga skupina so algoritmi, ki so bili implementirani le za namen testiranja prve skupine. Zanimala nas je torej uspešnost in sposobnost učenja agentov iz prve skupine.

Pričakovali smo, da se bodo agenti iz prve skupine izkazali zelo dobro proti agentom iz druge, saj so vzorci, ki jih le-ti tvorijo, precej preprosti. Manjša uspešnost je bila pričakovana proti drugim računalniško generiranim testnim nizom, saj so ti vsebovali manj razpoznavne vzorce, vendar smo vseeno pričakovali uspešnost nad 50%. Končno smo pričakovali še, da se bodo algoritmi iz prve skupine v spopadu med sabo izkazali dokaj izenačeno.

Delo je sestavljeno iz 5 poglavij. V 2. poglavju pregledamo že obstoječe implementacije igranja igre Kamen, papir, škarje in jih komentiramo. V 3. poglavju opišemo implementacije izbranih agentov. V 4. poglavju analiziramo rezultate in primerjamo uspešnost agentov. V 5. poglavju povzamemo celotno delo v diplomski nalogi in navedemo glavne ugotovitve. Predstavimo ideje za nadaljnje raziskave na tem področju.

Poglavje 2

Pregled področja

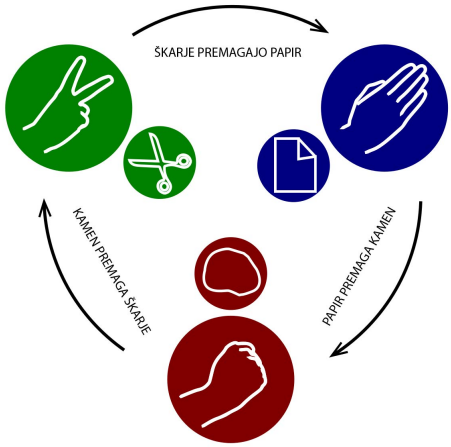
2.1 Pravile igre

Igra Kamen, papir, škarje, kuščar, Spock je preprosta igra za dva igralca, pri kateri igralca z roko tvorita določene oblike, vsaka od teh oblik pa je v naprej določenemu razmerju z ostalimi oblikami. V osnovni različici igre imata igralca na voljo tri oblike oziroma poteze: kamen (stisnjena pest), papir (iztegnjena dlan) in škarje (iztegnjena kazalec in sredinec). Zmagovalec vsake igre je določen na podlagi treh preprostih pravil:

1. Kamen premaga škarje.
2. Škarje premagajo papir.
3. Papir premaga kamen.

Pri razširjeni različici igre trem osnovnim potezam dodamo še dve: kuščar in Spock. Pravila se temu primerno prilagodijo:

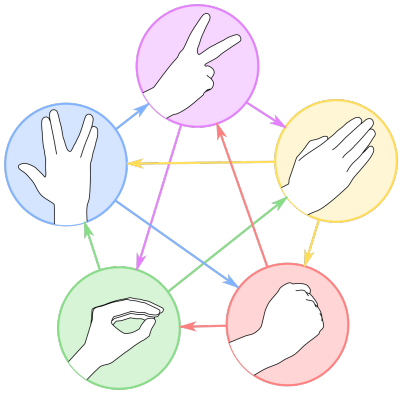
1. Kamen premaga škarje.
2. Kuščar premaga papir in Spocka.
3. Spock premaga škarje in kamen.
4. Škarje premagajo papir in kuščarja.



Slika 2.1: Razmerje med potezami pri igri Kamen, škarje, papir. [14]

5. Papir premaga kamen in Spocka.

Vsaka poteza je proti sebi vedno izenačena.



Slika 2.2: Razmerje med potezami pri igri Kamen, škarje, papir, kuščar, Spock. [12]

Opisana pravila so grafično ponazorjena na slikah 2.1 in 2.2.

Smisel igre je v tem, da igralca hkrati igrata vsak svojo potezo in nato razbereta zmagovalca [16].

2.2 Pregled sorodnih del

Medtem ko je področje igranja iger s pomočjo umetne inteligence precej aktivno in raziskovano področje, je večina pozornosti namenjena drugim, kompleksnejšim igram, kot na primer šah. Za reševanje problema napovedovanja potez pri igri Kamen, papir, škarje (ali njene razširitve) tako ne obstaja neka ustaljena metoda ali algoritem, za katero bi veljalo, da je najbolj uspešna. Navedeno velja seveda le za metode ali algoritme, ki pri napovedovanju potez ne uporabljajo nič drugega kot le zgodovino dosedanjih iger. S pomočjo dodatnih tehnik in pristopov pa je možno doseči uspešnost krepko nad 50%.

Na univerzi v Tokiu [3] so razvili robota za igranje igre Kamen, papir, škarje, ki lahko proti človeškemu igralcu doseže maksimalno uspešnost. Pri tem je ključna uporaba dodatnih vhodnih podatkov: robot opazuje gibanje nasprotnikove roke in prepozna njegovo naslednjo potezo. Na tak način je sposoben vedno odigrati potezo, s katero premaga nasprotnika. Človeški igralci pri igranju tvorijo svoje poteze v relativno istem času - nasprotnik nima dovolj časa, da bi prepoznal nasprotnikovo potezo in se ustrezno odzval. Robot teh težav seveda nima; ni le sposoben razbrati, kaj bo nasprotnik odigral, preden ta dejansko zaključi svojo potezo, temveč v tem kratkem intervalu (med točko, ko robot pravilno razpozna nasprotnikovo naslednjo potezo in točko, ko je nasprotnikova poteza dokončno izoblikovana) tvori tudi svojo potezo. Za človeški vid celoten dogodek izgleda kot poštena igra.

Druga skrajnost doseganja maksimalne uspešnosti so agenti, ki na nasprotnikovo potezo odreagirajo šele po tem, ko jo preberejo. Na spletni strani podjetja Afiniti se je namreč pojavila tako imenovana umetna inteligenca za igranje igre Kamen, papir, škarje in se ponašala z zelo dobrim razmerjem zmag in porazov proti človeškim igralcem, ki so igro igrali na njihovi strani. Na presenečenje Jamesa Stanleya, ki je izvedel manjšo raziskavo na omenjenem algoritmu, je algoritem odločno povedel tudi proti računalniško generiranim naključnim potezam [11]. To seveda pomeni, da algoritem ni pošten, saj proti naključnim potezam ne moremo napovedovati. Tak algo-

ritem pred igranjem svoje poteze že prebere nasprotnikovo potezo - če bi hoteli, bi lahko imeli 100% uspešnost, vendar potem ne bi izgledalo, kot da se algoritem uči in izboljšuje.

Primer poštenega algoritma, ki igra in se uči z metodami umetne inteligence, je implementiran v robotu iz Univerze v Padovi, kjer so posebno pozornost namenili posnemanju realne igre med dvema igralcema [5]. Za igranje proti robotu uporabniki ne potrebujejo daljinca ali tipkovnice, saj le-ta uporablja RGB-D senzor. Robot za strojno učenje uporablja metodo GMM (Gaussian Mixture Model) [6].

Algoritmi pa ne tekmujejo samo proti človeškim igralcem. Prvo mednarodno tekmovanje med algoritmi v igri Kamen, papir, škarje je potekalo leta 1999 in v njem je nastopilo 39 tekmovalcev iz 10 različnih držav [1]. Število oddanih algoritmov na igralca ni bilo omejeno, zato se je na tekmovanju pomerilo kar 45 algoritmov. Na tekmovanju se je predvsem izkazal algoritem poimenovan "Iocaine Powder", njegov avtor je Dan Egnor - zmagal je namreč na vseh 25 turnirjih, ki so bili organizirani na dogodku. Algoritem je uporabljal različne strategije in napovedne algoritme pri izbiri svoje poteze, njegov uspeh pa je porodil številne kopije in izboljšave, ki temeljijo na osnovni verziji algoritma. Nekatere izmed njih je moč najti na spletni strani, ki predstavlja platformo za tekmovanje med algoritmi [7].

Omenjeno tekmovanje je bilo osnova za nadaljnje raziskave. Leta 2005 je bil objavljen članek, v katerem so avtorji zgradili nov algoritem in ga testirali proti vsem algoritmom iz tekmovanja. Algoritem je bil izboljšana verzija "Beat Frequent Pick (BFP)"algoritma, ki izbira svoje poteze na podlagi štetja nasprotnikovih potez. Rezultati so pokazali, da razviti algoritem še vedno ni bil kos "Iocaine Powderju", vendar so dodatne izboljšave danega algoritma vsekakor možne [13].

Obstajajo tudi hevristične strategije igranja igre KPŠ, ki temeljijo na statističnih podatkih človeških igralcev oziroma na sami človeški psihologiji [15]. Uspešnost takega algoritma ne bi nikoli bila blizu 100%, lahko pa pričakujemo, da bi algoritem, ki bi deloval na podlagi statističnih podatkov in

človeških tendenc, premagoval človeške nasprotnike več kot 50% časa. Ideja implementacije takega algoritma je na primer s spodbujevanim učenjem [2]. Tak algoritem se v igri proti ostalim algoritmom ne bi izkazal najbolje, saj mu poznavanje človeških vzorcev igranja pri igri proti računalnikom ne bi pomagalo.

2.3 Komentarji in izbira metodologije

V diplomskem delu smo se omejili predvsem na igranje igre Kamen, papir, škarje, kuščar, Spock s pomočjo algoritmov, ki bi se napovedovanja nasprotnikovih potez učili na podlagi zgodovine dosedanjih potez. Zaradi želje po večji količini podatkov smo se odpovedali igri proti človeškemu igralcu, posledično uporaba dodatnih vhodnih podatkov, kot so na primer globalna statistika potez ali procesiranje gibov, ni možna. Samo po sebi umevno je tudi dejstvo, da smo se omejili le na poštene algoritme - tiste, ki pri svoji napovedi ne upoštevajo nasprotnikove trenutne poteze.

Na srečo obstaja preprost način za ugotavljanje poštenosti algoritma. To je naključna igra. Popolnoma vsak pošten algoritem bo proti naključni igri imel 50% uspešnost. Za zgoraj omenjeno "pravilo" obstaja izjema v primeru, da bi algoritem zaznal vzorce v generatorju naključnih števil, ki ga uporabljamo za generiranje naključnih potez - noben generator naključnih števil ni resnično naključen.

50% uspešnost je torej spodnja meja, ki smo jo želeli preseči z implementacijami inteligentnih algotimov za napovedovanje na podlagi zgodovine potez - želeli smo biti boljši od naključja. Obstajajo različne, bolj ali manj uspešne strategije in algoritmi za dosego tega cilja. Naštejmo nekaj najpreprostejših [4]:

- **Nespremenljiva poteza.**

  Agent bo preprosto vedno igral isto potezo, na primer kamen.

- **Štetje frekvenc.**

Algoritem šteje, kolikokrat je nasprotnik odigral vsako od možnih potez. Nato odigra tisto potezo, ki bo premagala najpogosteje igrano nasprotnikovo potezo.

- **Rotacija in anti-rotacija.**

  Rotacijo pri igri Kamen, papir, škarje si predstavljamo kot modularno seštevanje z modulom 3. V tabeli 2.1 vidimo, da vsaka rotacija poteze za 1 premaga prejšnjo potezo. Na tej osnovi lahko zasnujemo dva preprosta algoritma; prvi, ki bo vedno odigral rotacijo za 1 glede na prejšnjo potezo, in drugi, ki bo predvideval, da nasprotnik počne le-to, in bo zato odigral rotacijo za 2.

Tabela 2.1: Prikaz rotacij potez pri igri Kamen, papir, škarje.

poteza	rotacija za 0	rotacija za 1	rotacija za 2	rotacija za 3
K	K	P	Š	K
P	P	Š	K	P
Š	Š	K	P	Š

- **Metoda ujemanja preteklih nizov**

  Ta agent si shranjuje zgodovino potez (lahko le lastnih, le nasprotnikovih ali oboje) in posodablja trenutni ključ. Ključ je niz potez izbrane dolžine, recimo n, ki predstavljajo zadnjih n odigranih potez. Algoritem svojo potezo izbere tako, da se sprehodi skozi zgodovino vseh potez in išče podniz, ki se ujema s ključem. Nato razbere, katero potezo je nasprotnik po tistem zaporedju potez odigral takrat in predvideva, da bo to storil ponovno.

- **Spodbujevano učenje**

  Spodbujevano učenje lahko izvedemo tudi na podlagi zgodovine potez, ne le na podlagi vnaprej danih strategij [2]. V tem primeru algoritem preprosto izbira poteze: če se izkaže, da je njegova izbira pravilna, je

nagrajen, sicer je kaznovan. Na tak način se algoritem na dolgi rok nauči izbirati poteze, ki pogosteje zmagujejo.

- **Markovske verige**

  Ideja markovskih verig je v tem, da na podlagi trenutnega stanja ocenjujemo verjetnosti za naslednja stanja. Če za trenutno stanje vzamemo zadnjih nekaj potez, za naslednja stanja pa naslednjo potezo, lahko markovske verige uporabimo za napovedanje poteze na podlagi zgodovine potez.

Poudarek našega dela je na zadnjih treh strategijah. Vse podrobnosti o algoritmih in implementaciji so opisane v 3. poglavju, za potrebe raziskovanja učinkovitosti teh algoritmov pa smo implementirali tudi nekaj drugih, preprostejših strategij.

Poglavje 3

Opis implementacije

3.1 Splošno o implementaciji

Celotna koda za diplomsko delo je napisana v programskem jeziku C#. Program izvajamo prek ukazne vrstice, brez grafičnega vmesnika.

Vse možne poteze poimensko shranjujemo v C# tipu enum. Pri igranju proti testnim nizom, ki jih beremo iz tekstovne datoteke, poskrbimo za ustrezno pretvorbo znakov:

```
enum Move
{
    Rock,
    Paper,
    Scissors,
    Lizard,
    Spock
}
Move StringToMove(string s)
{
    return s switch
    {
        "r" => Move.Rock,
        "p" => Move.Paper,
        "s" => Move.Scissors,
        "L" => Move.Lizard,
        "S" => Move.Spock,
    };
```

```
}
```

Poskrbeli smo, da vsi algoritmi naključno izberejo ustrezno potezo glede na napovedano nasprotnikovo potezo:

```
Move GetCounterTo(Move move)
{
    Random r = new Random();
    switch (move)
    {
        case Move.Rock:
            Move[] countersRock = { Move.Paper, Move.Spock };
            return countersRock[r.Next(0, 2)];
        case Move.Paper:
            Move[] countersPaper = { Move.Scissors, Move.Lizard };
            return countersPaper[r.Next(0, 2)];
        case Move.Scissors:
            Move[] countersScissors = { Move.Rock, Move.Spock };
            return countersScissors[r.Next(0, 2)];
        case Move.Lizard:
            Move[] countersLizard = { Move.Rock, Move.Scissors };
            return countersLizard[r.Next(0, 2)];
        case Move.Spock:
            Move[] countersSpock = { Move.Lizard, Move.Paper };
            return countersSpock[r.Next(0, 2)];
    }
}
```

V nadaljevanju bomo z besedo "ključ" označevali niz dolžine r, ki predstavlja skupek odigranih potez - vsaka črka predstavlja ustrezno potezo. Z besedno zvezo "trenutni ključ" bomo označevali zadnjih r nasprotnikovih potez.

3.2 Markovske verige

Markovska veriga je stohastični model, ki opisuje zaporedje možnih dogodkov, pri katerem je verjetnost vsakega dogodka odvisna le od stanja prejšnjega dogodka. Verjetnostna porazdelitev prehoda med stanji je ponavadi podana s prehodno matriko. Markovska veriga z n možnimi stanji bo tako imela

prehodno matriko velikosti $n \times n$ tako, da element na mestu (i, j) predstavlja verjetnost prehoda iz stanja i v stanje j. Prehodna matrika je hkrati stohastična matrika, kar pomeni, da je vsota vsake vrstice natanko 1 [8].

Prilagoditve in implementacija

V diplomskem delu se nismo držali striktne matematične definicije markovskih verig, vzeli smo le idejo in jo preoblikovali glede na svoje potrebe. Izvedli smo dve glavni spremembi:

1. Prehodna matrika ne bo več kvadratna $n \times n$ matrika, temveč pravokotna matrika s nespremenljivo dolžino 5 vrstic; vsak stolpec predstavlja eno od petih možnih nasprotnikovih izbir. Število vrstic bo enako številu vseh možnih nizov dolžine r, sestavljenih iz treh znakov, r pa je število, ki določa, koliko prejšnjih potez hranimo. S parametrom r (v nadaljevanju tudi "red") torej določimo, koliko potez za nazaj pomnimo.

2. Prehodna matrika ne bo več stohastična, saj te lastnosti za napoved ne potrebujemo. Vedno bomo predvidevali, da bo nasprotnik odigral znak, ki ima v opazovani vrstici največjo vrednost.

Namesto matrike smo uporabili podatkovno strukturo slovar (angl. *dictionary*), saj v primerjavi z ostalimi podatkovnimi strukturami omogoča hitrejše iskanje po ključih. Slovar deluje tako, da si shranjuje unikatne ključe, vsakemu ključu pa priredi nek objekt. V našem primeru je ključ posamezen niz dolžine r, ki predstavlja prejšnjih r odigranih znakov s strani nasprotnika - vseh ključev je 5^r: imamo r mest, na vsako mesto lahko vpišemo eno od petih črk: 'r' - rock, 'p' - paper, 's' - scissor, 'L' - lizard ali 'S' - Spock, ki predstavljajo poteze nasprotnika. Takšen slovar smo v jeziku C# generirali tako:

```
PrepareDictionary(int order)
{
    _allKeys = new List<string>();
    string masterString = "rpsLS";
    string currentKey = new string('r', order);
```

```
    GetAllKeys(masterString, currentKey, order, 0);
    foreach (string s in _allKeys)
    {
        _dictionary.Add(s, new int[5]);
    }
}
GetAllKeys(string masterString, string currentKey, int order, int index)
{
    if (index >= order)
    {
        _allKeys.Add(currentKey);
    }
    else
    {
        foreach (char c in masterString)
        {
            string beg = currentKey.Substring(0, index);
            string end = currentKey.Substring(index + 1, currentKey.Length
                - index - 1);
            string newKey = beg + c + end;
            GetAllKeys(masterString, newKey, order, index + 1);
        }
    }
}
```

Vsakemu ključu smo priredili seznam s petimi celoštevilskimi vrednostmi, ki predstavljajo možne poteze in so na začetku nastavljene na 0 - igra se še ni začela, nobena poteza še ni bila odigrana. Po dogovoru prvi element seznama šteje število odigranih kamnov, drugi element število odigranih papirjev, tretji število odigranih škarij, četrti število odigranih kuščarjev in peti število odigranih Spockov.

Algoritem

Algoritem v psevdokodi izgleda tako:

```
InitializeDictionary();
while (ContinuePlaying())
{
        if (numberOfGames < r + 1)
        {
                PlayRandom();
```

```
        }
        else
        {
                PredictAndPlayAccordingly();
        }
        Learn(ReadOpponentsMove());
}
```

Dokler ne bo odigranih vsaj $r + 1$ iger, bo agent izbiral naključne poteze, saj nima dovolj podatkov za predvidevanje. Nato (po vsaj $r + 1$ odigranih igrah) bo lahko s trenutnim ključem dolžine r dostopal do slovarja in si na ustreznem mestu označil, kaj je igral nasprotnik.

Za zgled denimo, da imamo $r = 2$, da je nasprotnik za prejšnji dve potezi obakrat odigral papir in sedaj spet igra papir. Torej imamo trenutni ključ $'pp'$. V slovarju poiščemo seznam s ključem $'pp'$ in povečamo števec papirjev za 1 ($\{...'pp' : [0, 1, 0, 0, 0]...\}$). Od igre $r + 1$ naprej pa bo algoritem lahko tudi predvideval: s trenutnim ključem, ki predstavlja prejšnjih r potez, bo pogledal v slovar in predvidel najverjetneje igrani nasprotnikov znak - tisti, z največjim števcem, nato pa bo zase izbral znak, ki premaga predvidenega. V primeru, da bo dani seznam imel več enakih vrednosti, se bo agent naključno odločil za eno izmed njih. V primeru ključa $'pp'$ bo torej prebral zapis $\{...'pp' : [0, 1, 0, 0, 0]...\}$ in predvidel, da je naslednja nasprotnikova poteza najverjetneje papir, zato bo odigral na primer s škarjami.

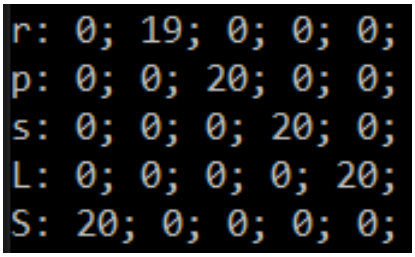
Nato bo, enako kot prej, povečal števec *dejansko* odigranega znaka ter ustrezno posodobil trenutni ključ.

Na sliki 3.1 vidimo markovsko matriko reda 1 po 100 odigranih igrah proti algoritmu, ki vnedogled izmenjuje poteze v istem vrstnem redu.

3.3 Metoda ujemanja preteklih nizov

Prilagoditve in implementacija

Implementirali smo metodo ujemanja preteklih nizov, ki si shranjuje le poteze nasprotnika in poišče zadnji pojav trenutnega ključa. Trenutni ključ



Slika 3.1: Markovska matrika odraža nasprotnikovo strategijo.

predstavlja r najnovejših nasprotnikov potez. Nato sklepa, da bo nasprotnik odigral isto potezo, kot jo je igral prejšnjič po tem zaporedju.

Osnovni algoritem, kot smo ga opisali v 2 poglavju, se je izkazal za prepočasnega pri učenju na večjem številu potez. Namesto shranjevanja celotne zgodovine iger smo zato generirali vse možne nize dolžine r (na enak način, kot pri markovskih verigah). Shranili smo jih v slovar kot ključe in vsakemu ključu priredili natanko eno vrednost - ta je predstavljala nasprotnikovo potezo, ki jo je nazadnje odigral po danem zaporedju, kot ga navaja ustrezni ključ.

Na tak način smo časovno zahtevnost iskanja ustreznega niza prevedli z $O(n^2)$ na $O(1)$ in pohitrili algoritem na račun prostorske zahtevnosti, vendar je situacija za majhne ključe povsem obvladljiva.

Algoritem

Algoritem v psevdokodi izgleda tako:

```
InitializeDictionary();
while (ContinuePlaying())
{
        if (numberOfGames < r + 1)
        {
                PlayRandom();
        }
        else
        {
                PredictAndPlayAccordingly();
        }
```

```
        Learn(ReadOpponentsMove());
}
```

Dokler ne bo odigranih vsaj $r + 1$ iger, bo agent izbiral naključne poteze, saj nima dovolj podatkov za predvidevanje. Nato (po vsaj $r + 1$ odigranih igrah) bo lahko s trenutnim ključem dolžine r dostopal do slovarja in si na ustreznem mestu označil, kaj je igral nasprotnik.

Za zgled denimo, da imamo $r = 1$, da je nasprotnik za prejšnjo potezo odigral kamen in da sedaj odigra papir. Torej imamo trenutni ključ $'k'$. V slovarju poiščemo potezo z danim trenutnim ključem in sklepamo: ker je nasprotnik nazadnje igral to potezo po tem, ko je prej odigral kamen, bo to storil ponovno. Nato za svojo potezo izberemo potezo, ki premaga predvideno nasprotnikovo potezo.

Po odigrani igri bo algoritem staro potezo nadomestil z dejansko odigrano nasprotnikovo potezo in ustrezno posodobil trenutni ključ.

3.4 Spodbujevano učenje na podlagi potez

V kontekstu umetne inteligence je spodbujevano učenje tip dinamičnega programiranja, pri katerem se algoritem oziroma agent uči iz zaporedja kazni in nagrad. Agent nagrado prejme, če se vede pravilno, v nasprotnem primeru pa prejme kazen. Agent se uči pravilnega vedenja tako, da poskuša minimizirati kazni ter maksimizirati nagrade [10].

Prilagoditve in implementacija

Spodbujevano učenje je mogoče implementirati na več različnih načinov. Za dani problem smo se odločili za implementacijo s pomočjo tako imenovanega Q-učenja. Q-učenje je izraz za strukturo algoritma, ki za spodbujevano učenje uporablja agenta, definira množico vseh možnih stanj in množico vseh možnih akcij za vsako stanje [9].

Množico vseh možnih stanj smo definirali kot množico vseh možnih nizov dolžine r, kjer vsak znak v nizu predstavlja eno od petih možnih potez. Moč

te množice je torej 5^r. Z drugimi besedami, to je množica vseh možnih ključev dolžine r. Množico vseh možnih akcij za vsako stanje smo definirali pri vsakem stanju enako: to je množica vseh petih različnih možnih potez. Vsaka akcija ima številsko vrednost, ki predstavlja nagrado oziroma kazen, ki jo agent prejme, če izbere to akcijo.

Definirali smo še naslednje parametre, ki so potrebni za Q-učenje:

- hitrost učenja - določa pomembnost pridobljenih informacij v primerjavi s starimi informacijami (pri vrednosti 1 bo agent ignoriral prej pridobljeno znanje, pri vrednosti 0 pa se ne bo učil).

- popust (angl. $discount factor$) - določa pomembnost prihodnjih nagrad (pri vrednostih blizu 0 bo agent kratkoviden, pri vrednostih blizu 1 bo pa stremel k dolgoročnim nagradam).

- ϵ - prag. Algoritem pred vsako potezo generira naključno število med 0 in 1. Če je generirano število večje od ϵ, algoritem odigra naključno.

V pomoč pri implementaciji spodbujevanega učenja s Q-učenjem nam je bil razred QTable, ki smo ga definirali sami: glaven atribut tega razreda razreda je slovar, ki predstavlja matriko nagrad in kazni. Razred vsebuje še pomožne metode za operacije nad tem slovarjem. Ta slovar bomo v nadaljevanju pogosto imenovali kar Q-tabela. Zaradi potrebe dostopanja do Q-tabele agent hrani in posodablja tudi trenutni ključ.

Vrstice v Q-tabeli so torej ključi iz množice vseh možnih stanj. Vrstice so dolžine 5, saj nam stolpci predstavljajo vsako od možnih akcij - potez. Po dogovoru nam prvi stolpec predstavlja trenutno nagrado oziroma kazen v primeru, da izberemo kamen, drugi stolpec nam na enak način predstavlja vrednost v primeru izbire papirja, tretji nam predstavlja vrednost za škarje, četrti in peti pa za kuščarja in Spocka.

Vrednosti v Q-tabeli živijo na intervalu od -1 do 1, pri čemer negativne vrednosti razumemo kot kazen, pozitivne pa kot nagrado. Ob začetni inicializaciji Q-tabele vse vrednosti nastavimo naključno.

Algoritem

Algoritem v psevdokodi izgleda tako:

```
PrepareQTable();
while (ContinuePlaying())
{
        if (numberOfGames < r)
        {
                PlayRandom();
        }
        else
        {
                TakeActionFromQTable();
        }
        Learn(ReadOpponentsMove());
}
```

Struktura algoritma je zelo podobna prejšnjim, možno pa je opaziti eno razliko. Agent preneha igrati naključno že pri $r-ti$ potezi, vsi prejšnji agenti pa šele pri $(r+1)-ti$. Ta razlika je v resnici le navidezna, saj tudi če ta agent s spobujevanim učenjem $r-to$ potezo odigra na podlagi Q-tabele, bo le-ta vsebovala samo naključno inicializirane vrednosti. Isto bi bilo, če bi pri $r-ti$ potezi poskušali predvidevati s pomočjo markovskih verig: našli bi seznam s petimi ničlami, in izbrali naključno. Podrobneje si oglejmo, kako agent s Q-učenjem izbere potezo in kako se uči.

Agent v metodi $TakeActionFromQTable()$ najprej izračuna naključno vrednost na intervalu od 0 do 1 in jo primerja s podanim parametrom ϵ. Če je računana vrednost večja od ϵ, potem agent odigra kar z naključno potezo. V nasprotnem primeru pa si pomaga s Q-tabelo. Naključno obnašanje je del Q-učenja in je potrebno, da agent ne bi obtičal v "lokalnem optimumu". V kontekstu igre Kamen, papir, škarje, kuščar, Spock si lokalni optimum lahko predstavljamo kot neko zelo dobro strategijo, vendar ne najboljšo možno. Agent bo kar naprej igral to strategijo, saj ga zadostno nagrajuje, zato nikoli ne bo našel (naj)boljše strategije.

To lahko predstavimo tudi na konkretnemu primeru. Denimo, da igramo proti računalniško generiranemu nizu, ki vnedogled ponavlja zaporedje štirih

kamnov, ki jim sledi en papir. Agent s spodbujevanim učenjem bi se lahko naučil, da je dobro vedno igrati papir, saj bo tako vedno zmagal oziroma izenačil. Če pa ga prisilimo, da določen odstotek iger odigra naključno, bo morda ugotovil, da je boljše odigrati štirikrat papir, nato pa škarje.

V primeru, da agent ne bo odigral naključno, bo dostopal do Q-tabele s trenutnim ključem in iz dane vrstice odčital največjo vrednost. V primeru enakih vrednosti bo izbral naključno eno izmed njih. Poteza, ki ustreza največji vrednosti, se interpretira kot poteza z največjo nagrado, zato jo agent izbere zase.

Sledi faza učenja, v kateri bo agent le posodobil ustrezno vrednost v Q-tabeli.

```
UpdateQTable(int reward, string oldState, string newState, int lastMove)
{
    _qTable[oldState][lastMove] = _qTable[oldState][lastMove] +
        _learningRate*(reward + _discount*_qTable[newState].Max() -
        _qTable[oldState][lastMove]);
}
```

Pomen spremenljivk *_discount* in *_learningRate* že poznamo; gre za vhodna parametra popust in hitrost učenja. Spremenljivka *reward* zavzame vrednost 1 v primeru, da je agent zmagal, v primeru poraza je -1, v primeru izenačenja pa je 0. Spremenljivki *oldState* in *newState* sta potrebni za orientacijo v Q-tabeli, predstavljata pa stari ključ, s katerim je agent odigral prejšnjo igro ter novi ključ, pridobljen iz starega, ko mu je agent dodal najnovejšo nasprotnikovo potezo. Spremenljivka *lastMove* predstavlja agentovo nazadnje odigrano potezo.

Funckija *UpdateQTable*() torej vzame vrednost, ki jo je agent prej izbral, in jo posodobi, v poštev pa vzame vse zgoraj naštete parametre ter spremenljivke. Pri izračunu upošteva tudi največjo vrednost iz vrstice v Q-tabeli, ki jo določa novi trenutni ključ *newState*.

3.5 Meta-klasifikator s spodbujevanim učenjem

Meta-klasifikator je v našem primeru algoritem, ki namesto same poteze izbira med drugimi algoritmi. Poteza, ki jo bo napovedal algoritem, izbran s strani meta-klasifikatorja, bo hkrati tudi poteza meta-klasifikatorja.

Prilagoditve in implementacija

Odločili smo ze, da bomo za pomožne algoritme vzeli kar vse implementirane algoritme. Še naprej smo uporabljali Q-učenje, zato smo ustrezno redefinirali določene strukture.

Množica vseh stanj je ohranila svoj pomen; to je množica vseh možnih ključev dolžine r, množica vseh možnih akcij pa se je spremenila: akcija več ni predstavljala ene od možnih potez, temveč enega od možnih agentov, ki bo potezo izbral sam. Vsaka akcija ima številsko vrednost, ki predstavlja nagrado oziroma kazen, ki jo agent prejme, če izbere to akcijo. Nagrajujemo oziroma kaznujemo torej izbiro agentov - če bo izbran agent uspešen, bo meta-klasifikator nagrajen, sicer bo kaznovan.

Definirali smo iste parametre, ki so potrebni za Q-učenje:

- hitrost učenja - določa pomembnost novo pridobljenih informacij v primerjavi s starimi informacijami (pri vrednosti 1 bo agent ignoriral prej pridobljeno znanje, pri vrednosti 0 pa se ne bo učil).
- popust - določa pomembnost prihodnjih nagrad (pri vrednostih blizu 0 bo agent kratkoviden, pri vrednostih blizu 1 bo pa stremel k dolgoročnim nagradam).
- ϵ - prag. Algoritem pred vsako potezo generira naključno število med 0 in 1. Če je generirano število večje od ϵ, algoritem odigra naključno.

Posodobili smo razred QTable in ga preimenovali v QTableAlgo. Vrstice Q-tabele, ki predstavlja matriko nagrad in kazni, so ohranile svoj pomen - ključi iz množice vseh možnih stanj, spremenila pa se je njihova dolžina, saj

nam stolpci sedaj predstavljajo vsakega od možnih agentov. Teh je 9, taka je tudi dolžina vrstic. Z dogovorom določimo, kako vrstni red stolpcev določa, katerega od agentov predstavlja vsak od njih.

Vrednosti v Q-tabeli ostajajo realna števila na intervalu od -1 do 1, pri čemer negativne vrednosti razumemo kot kazen, pozitivne pa kot nagrado. Ob začetni inicializaciji Q-tabele vse vrednosti nastavimo naključno.

Algoritem

Psevdokoda algoritma je identična psevdokodi agenta s spodbujevanim učenjem na podlagi potez, dejanska implementacija pa ima določene posebnosti.

Naprednejši pomožni agenti, kot na primer agent z markovskimi verigami, napovedujejo poteze na podlagi zgodovine iger. Agenti potrebujejo celotno zgodovino iger da bi bile napovedi uspešne, zato meta-klasifikator skrbi za izvajanje - igranje in učenje - vseh svojih pomožnih agentov, čeprav izbere le enega. Izbira pomožnega agenta poteka na podlagi vrednosti iz Q-tabele, identično kot pri agentu s spodbujevanih učenjem na podlagi potez.

Posebno skrb smo namenili agentu s spodbujevanim učenjem na podlagi potez, saj je izmed pomožnih agentov edini, ki kot parameter za učenje potrebuje tudi svojo zadnjo odigrano potezo. Smiselno je, da se uči na podlagi svoje lastne poteze, tudi če ta ni bila izbrana s strani meta-klasifikatorja. S tem je poskrbljeno za ustrezno posodabljanje nagrad in kazni v Q-tabeli - tudi, če je meta-klasifikator izbral agenta, ki je izbral zmagovalno potezo, se mora agent s spodbujevanim učenjem kaznovati v primeru, če bi njegova izbrana poteza izgubila. Pri ostalih agentih ta dodatek ni bil potreben, saj se učijo le na podlagi nasprotnikove poteze, neodvisno od svoje lastne poteze.

3.5.1 Meta-klasifikator z izbranimi algoritmi

Zadnji agent, ki smo ga implementirali, je identičen meta-klasifikatorju z vsemi agenti, le da podpira le izbrane tri algoritme: metoda ujemanja preteklih nizov, markovske verige in algoritem s spodbujevanim učenjem na podlagi potez. Namerno smo izbrali le tri agente, saj smo od njih pričakovali

najboljšo uspešnost. Zanimalo nas je, ali bo ta meta-klasifikator boljši od prvotnega in koliko.

3.6 Pomožni algoritmi

Pomožni algoritmi so namenjeni testiranju glavnih algoritmov in so kot taki zelo preprosti. Implementirali smo naslednje pomožne algoritme:

- **Vedno kamen**

  Vedno kamen je algoritem, ki vedno vrne kamen. Zanimiv je predvsem za testiranje, kako hitro se dani agent lahko nauči tega preprostega vzorca.

- **Vedno obračaj**

  Vedno obračaj je algoritem, ki vedno igra v enakem zaporedju: kamen, papir, škarje, kuščar, Spock, kamen, in tako naprej. Gre za zahtevnejši vzorec. Zanimivo je, da se algoritem iz članka o modificiranem BFP algoritmu ni naučil igrati proti algoritmu vedno obračaj [13].

- **Vedno premagaj**

  Vedno premagaj je algoritem, ki bo vedno odigral potezo, ki premaga nasprotnikovo prejšnjo potezo.

- **Vedno premagaj sebe**

  Algoritem vedno premagaj sebe vedno odigra potezo, ki premaga svojo lastno prejšnjo potezo.

Poglavje 4

Evalvacija in rezultati

4.1 Algoritmi proti algoritmom

4.1.1 Opis eksperimenta

Algoritmi so v eksperimentih nastopali v dveh vlogah: vloga preučevanega algoritma in vloga testnega algoritma. Preučevani algoritem je tekmoval proti testnemu algoritmu.

Ker smo želeli testirati tudi vpliv vrednosti parametrov pri preučevanem algoritmu, smo tekmo preučevanega algoritma proti testnim algoritmom ponovili večkrat, vsakič z drugačnimi vrednostmi parametrov. Vsaka tekma je bila sestavljena iz 100.000 potez, algoritmi pa niso hranili znanja iz prejšnjih tekem. Algoritem v vlogi testnega algoritma je imel vedno iste vrednosti svojih parametrov, imenovali smo jih privzeti parametri.

Pregled preučevanih in privzetih parametrov

Metoda ujemanja preteklih nizov. Najbolj zanimiv preučevan parameter pri metodi ujemanja preteklih nizov je dolžina ujemajočega niza. Odločili smo se za dolžine od 1 do 6, privzeto dolžino pa smo nastavili na 2.

Markovske verige. Pri markovskih verigah smo preučevali red markovskih verig. Izbrali smo rede od 1 do 6, privzeti red pa je bil 3.

Spodbujevano učenje na podlagi potez. Vsi algoritmi s spodbujevanim učenjem na podlagi Q-učenja so določeni s štirimi parametri:

1. Red. Preučevali smo rede od 1 do 6.
2. Hitrost učenja. Preučevano in privzeto hitrost učenja smo nastavili na 0.2.
3. Popust. Preučevan in privzeti popust smo nastavili na 0.9.
4. ϵ. Preučevali smo dve različne vrednosti parametra: 0.9 in 0.95, medtem ko je privzeti ϵ bil 0.9.

Meta-klasifikator s spodbujevanim učenjem. Poleg zgoraj opisanih parametrov za spodbujevano učenje smo določili še algoritme, med katerimi meta-klasifikator izbira. Le-ti so bili vedno isti; uporabili smo kar vse implementirane algoritme, tudi pomožne.

Pri meta-klasifikatorju z izbranimi algoritmi smo uporabili le tri algoritme: metoda ujemanja preteklih nizov, markovske verige in algoritem s spodbujevanim učenjem na podlagi potez.

4.1.2 Pregled rezultatov

Rezultate celotnega tekmovanja med algoritmi prikazuje tabela 4.1, v tabeli 4.2 pa so zbrane povprečne uspešnosti preučevanih algoritmov pri tekmovanju proti testnim algoritmom. Povprečja so izračunana preko vseh različnih nastavitev preučevanih parametrov, zaradi lepšega prikaza pa so upoštevane le igre, ki se niso končale izenačeno - 50% uspešnost torej pomeni, da je algoritem zmagal 33% vseh iger.

Opazili smo, da sta algoritma markovske verige in metoda ujemanja preteklih nizov proti najbolj predvidljivim pomožnim algoritmom (vedno obračaj

	MUPN	MV	SUPP	MKSU	MKIA
VP	50.00%	60.00%	58.00%	58.00%	60.00%
VPS	86.00%	86.00%	94.00%	93.00%	94.00%
VO	100.00%	100.00%	98.00%	99.00%	100.00%
VK	100.00%	100.00%	98.00%	100.00%	100.00%
MUPN	50.00%	50.00%	49.00%	50.00%	50.00%
MK	50.00%	50.00%	50.00%	50.00%	50.00%
Naključje	50.00%	50.00%	50.00%	50.00%	50.00%
SUPP	50.00%	50.00%	49.00%	50.00%	50.00%
MKSU	50.00%	49.00%	49.00%	50.00%	50.00%

Tabela 4.1: Rezultati tekmovanja med algoritmi.

in vedno kamen) dosegla maksimalno uspešnost, medtem ko sta algoritma s spodbujevanim učenjem obtičala pri 98% - 99% uspešnosti, meta-klasifikator z izbranimi algoritmi pa, čeprav temelji na spodbujevanem učenju, dosega 100% uspešnost. Razlog za to je v dejstvu, da metoda ujemanja preteklih nizov in markovske verige ne vsebujeta faktorja naključnosti; vedno igrata z naučenim vzorcem, ki se v primeru teh pomožnih algoritmov nikoli ne spreminja. Algoritma s spodbujevanim učenjem pa sta prisiljena določen del časa izbrati nekaj naključnega in lahko se zgodi, da izbrana poteza izgubi. Pri meta-klasifikatorju z izbranimi algoritmi tudi lahko pride do naključnega obnašanja, vendar imata dva od treh pomožnih algoritmov (markovske verige in metoda ujemanja preteklih nizov) 100% uspešnost, zato vseeno zmagamo, če ju izberemo. Delež iger, ki ga meta-klasifikator z izbranimi algoritmi izgubi zaradi naključnega obnašanja, je torej zelo majhen in se izgubi v zaokroževanju pri računanju povprečja.

ALGORITEM	POVPREČNA USPEŠNOST
MUPN	65.11%
MV	66.11%
SUPP	66.11%
MKSU	66.56%
MKIA	67.11%

Tabela 4.2: Povprečna uspešnost algoritmov pri tekmovanju proti ostalim algoritmom.

Rezultati, strnjeni na tak način, kažejo, da se je od vseh algoritmov v boju proti vsem ostalim najbolje izkazal meta-klasifikator z izbranimi algoritmi,

kar se ujema z našimi pričakovanji. Pričakovali smo, da bo meta-klasifikator s spodbujevanim učenjem boljši od svojih pomožnih algoritmov, prav tako pa smo pričakovali, da bo meta-klasifikator z izbranimi algoritmi boljši od njega. Razlika v uspešnosti je manjša, kot smo se nadejali, kar pomeni, da je že prvotni meta-klasifikator (torej tisti, ki izbira med vsemi algoritmi) pogosto izbiral najboljše tri algoritme.

Opazimo lahko tudi, da med algoritmi ni veliko odstopanj; med najboljšim in najslabšim povprečjem je le 2% razlike. Razlog za tako sliko podatkov je v tem, da so preučevani algoritmi med seboj zelo izenačeni (vsi rezultati te kategorije kažejo uspešnost 49% - 50% med katerimakoli dvema algoritmoma), po drugi strani pa so zelo uspešni proti pomožnim algoritmom (visoke uspešnosti tudi do 100%, na primer proti algoritmu vedno kamen), kot lahko vidimo v tabeli 4.3. Velike razlike med dvemi skupinami testnih algoritmov "prevladajo" nad manjšimi razlikami v posameznih bojih, zato skupno povprečje izpade navidez dokaj izenačeno.

PREUČEVANI ALGORITEM	TESTNI ALGORITEM	POVPREČNA USPEŠNOST
Markovske verige	VP	60.00%
	VPS	86.00%
	VO	100.00%
	VK	100.00%
	MUPN	50.00%
	MK	50.00%
	SUPP	50.00%
	MKSU	49.00%

Tabela 4.3: Povprečna uspešnost markovskih verig proti ostalim algoritmom.

Pri meta-klasifikatorju s spodbujevanim učenjem smo opazili povečanje uspešnosti proti pomožnemu algoritmu vedno premagaj v primeru, ko smo tekmovali z večjim parametrom ϵ (0.95 namesto 0.90). Rezultati so zbrani v tabeli 4.4.

Manjše okno za izbiro naključne izbire je torej porodilo boljše rezultate, iz česar lahko sklepamo, da se je v kontekstu igre KPŠLS bolje močno držati naučenih vzorcev, ki so že sami po sebi zelo šibki, kot pa raziskovati nove možnosti, v kolikor igramo proti predvidljivim algoritmom. Pomožni algori-

	Meta-klasifikator s spodbujevanim učenjem	
	$\epsilon = 0.90$	$\epsilon = 0.95$
TESTNI ALGORITEM		
Vedno premagaj	58.00%	60.00%

Tabela 4.4: Povprečna uspešnost meta-klasifikatorja proti algoritmu vedno premagaj pri različni vrednosti parametra ϵ.

tem vedno premagaj je namreč zelo predvidljiv algoritem, saj vedno odigra potezo, ki premaga nasprotnikovo prejšnjo potezo, zato naključno igranje proti njemu pokvari naučene vzorce in s tem uspešnost.

4.2 Algoritmi proti generiranim nizom

4.2.1 Opis eksperimenta

Zaradi hitrejšega izvajanja smo si v igri proti nizom privoščili tekme z milijonom potez. Nize potez smo generirali vnaprej, zato je vsak algoritem igral proti istim nizom. Privzetih parametrov v tem sklopu eksperimentov seveda ni bilo, preučevani parametri pa so enaki, kot so bili v prejšnjem eksperimentu.

Naključni nizi

S privzetim generatorjem naključnih števil iz programskega jezika C# smo generirali 10 naključnih nizov potez. Algoritmi so nato igrali proti tem nizom, pri čemer smo ustrezno spreminjali vrednosti preučevanih parametrov.

Test proti naključnim nizom smo izvedli zaradi preverjanja poštenosti algoritmov - pričakovali smo izenačene rezultate.

Nizi s fiksno preferenco

Naključnost nizov smo delno pokvarili tako, da smo na vsakih nekaj potez vrinili fiksno potezo: kamen. Generirali smo tri skupine nizov, ki kamen vrivajo na vsakih 3, 10 oziroma 100 potez. Za vsako skupino smo generirali

10 nizov. Algoritmi so se pomerili proti vsem tem nizom, pri čemer smo ustrezno spreminjali vrednosti preučevanih parametrov.

Ta sklop eksperimentov je bil izveden z namenom testiranja, kako dobro lahko preučevani algoritmi razpoznajo šibke, ponavljajoče vzorce.

Nizi z verjetnostno preferenco

Namesto fiksnega vrivanja potez smo v tem sklopu eksperimentov naključnost nizov pokvarili tako, da smo simulirali pristrani naključni generator. To pomeni, da generator ni več generiral vsake izmed petih potez z verjetnostjo 20%, ampak smo verjetnost za kamen variirali, medtem ko so ostale štiri verjetnosti bile med seboj enake. Generirali smo nize z verjetnostjo za kamen od 10% do 90% s skokom po 10%. Algoritmi so se pomerili proti vsem tem nizom, pri čemer smo ustrezno spreminjali vrednosti preučevanih parametrov.

Testiranje nizov z verjetnostno preferenco je zanimivo predvsem zaradi mnenja, da generiranje nizov na tak način še najbolj spominja na človeško igro.

4.2.2 Pregled rezultatov

Nizi s fiksno preferenco

Igra proti naključnim nizom je pokazala pričakovano 50% uspešnost. Algoritmi niso bili sposobni razpoznati vzorca v naključnem generatorju števil.

Še slabše, algoritmi niso bili sposobni razpoznati vzorca v nizih s fiksno preferenco za kamen na vsakih 100 potez. Rezultati, vidni iz tabele 4.5, so popolnoma enaki tistim, kot smo jih dobili pri igri proti naključnim nizom. Edina razlika je 51% uspešnost markovskih verig prvega reda, kar pa pripisujemo naključju.

Proti nizom s fiksno preferenco za kamen na vsakih 10 potez pa je že opazna premoč algoritmov. Kot kažejo rezultati iz tabele 4.6, so se najbolj izkazale markovske verige. Meta-klasifikator z izbranimi algoritmi se je spet izkazal boljše od osnovnega meta-klasifikatorja, kar se sklada z našimi

RED	MUPN	MV	SUPP	MKSU	MKIA
1	50.00%	51.00%	50.00%	50.00%	50.00%
2	50.00%	50.00%	50.00%	50.00%	50.00%
3	50.00%	50.00%	50.00%	50.00%	50.00%
4	50.00%	50.00%	50.00%	50.00%	50.00%
5	50.00%	50.00%	50.00%	50.00%	50.00%
6	50.00%	50.00%	50.00%	50.00%	50.00%

Tabela 4.5: Algoritmi proti nizom s fiksno preferenco za kamen na vsakih 100 potez.

pričakovanji. Ker so se markovske verige izkazale boljše od obeh meta-klasifikatorjev pa lahko sklepamo, da je sama struktura markovskih verig najbolj primerna za napovedovanje potez pri takem tipu nizov, zato je občasna odločitev meta-klasifikatorjev, ki so izbrali nek drug algoritem, poslabšala njihovo uspešnost.

Predvsem zanimivo je opažanje iz tabele 4.7, da pri višjih redih algoritma uspešnost pade. Razlog za to je v kombinatorični eksploziji števila vrstic v matriki. Število vrstic se veča eksponentno z večanjem reda. Ker vsaka vrstica v matriki reda r predstavlja točno določeno zaporedje r potez, se pri večjem redu to zaporedje manjkrat pojavi. Manj pojavitev pa pomeni slabše razpoznavanje vzorcev in s tem manjšo uspešnost. Za primer si lahko pogledamo razliko med redoma 4 in 6. Pri redu 4 imamo 625 vrstic - zaporedij, kar pomeni, da se vsako zaporedje pojavi približno 1600-krat. Pri redu 6 pa imamo 15625 vrstic in število pojavitev enega zaporedja se zmanjša na 64.

ALGORITEM	POVPREČNA USPEŠNOST
MUPN	50.83%
MV	55.33%
SUPP	52.17%
MKSU	53.67%
MKIA	53.83%

Tabela 4.6: Povprečna uspešnost algoritmov proti nizom s fiksno preferenco za kamen na vsakih 10 potez.

Tekmovanje algoritmov proti nizom s fiksno preferenco za kamen na vsake tri poteze je, po pričakovanjih, le še povečalo premoč algoritmov, ni pa obrodilo nobenih novih zanimivih vzorcev, ki jih nismo analizirali že pri nizih s

RED	MUPN	MV	SUPP	MKSU	MKIA
1	50.00%	56.00%	52.00%	54.00%	54.00%
2	51.00%	56.00%	52.00%	54.00%	54.00%
3	51.00%	56.00%	53.00%	54.00%	54.00%
4	51.00%	56.00%	53.00%	54.00%	54.00%
5	51.00%	55.00%	52.00%	53.00%	54.00%
6	51.00%	53.00%	51.00%	53.00%	53.00%

Tabela 4.7: Algoritmi proti nizom s fiksno preferenco za kamen na vsakih 10 potez.

preferenco za kamen na vsakih 10 potez. Povprečne uspešnosti in rezultati, razdeljeni po redih, so predstavljeni v tabelah 4.8 in 4.9.

ALGORITEM	POVPREČNA USPEŠNOST
MUPN	65.00%
MV	69.00%
SUPP	66.67%
MKSU	66.67%
MKIA	67.83%

Tabela 4.8: Povprečna uspešnost algoritmov proti nizom s fiksno preferenco za kamen na vsake tri poteze.

RED	MUPN	MV	SUPP	MKSU	MKIA
1	57.00%	69.00%	67.00%	67.00%	67.00%
2	65.00%	69.00%	67.00%	67.00%	68.00%
3	66.00%	69.00%	68.00%	68.00%	68.00%
4	67.00%	69.00%	67.00%	68.00%	68.00%
5	68.00%	69.00%	65.00%	67.00%	68.00%
6	68.00%	69.00%	66.00%	67.00%	68.00%

Tabela 4.9: Algoritmi proti nizom s fiksno preferenco za kamen na vsake tri poteze.

Nizi z verjetnostno preferenco

V tekmi proti nizom z verjetnostno preferenco je ponovno dominiral algoritem z markovskimi verigami, kot lahko vidimo iz tabele 4.10. Povprečja so izračunana po vseh različnih preučevanih parametrih ter po vseh različnih verjetnostih za kamen - te so tekle od 10% do 90% s skokom po 10%. Poleg dominance markovskih verig se je ohranil tudi že ustaljeni trend prevlade

meta-klasifikatorja z izbranimi algoritmi nad osnovno različico, kar je bilo po pričakovanjih.

ALGORITEM	POVPREČNA USPEŠNOST
MUPN	63.56%
MV	71.22%
SUPP	67.44%
MKSU	68.89%
MKIA	69.11%

Tabela 4.10: Povprečna uspešnost algoritmov proti nizom z verjetnostno preferenco za kamen.

Zelo visoka uspešnost algoritmov proti nizom z veliko verjetnostno preferenco za kamen sama po sebi ni impresivna. Denimo, da imamo agenta, ki vedno igra papir. Opazili bi, da tudi ta agent dosega skoraj enake uspešnosti kot naši preučevani algoritmi. Uspešnost naših algoritmov je torej potrebno evalvirati v primernem kontekstu: koliko so preučevani algoritmi boljši od agenta, ki vedno igra papir.

Od agenta, ki vedno igra papir, lahko pričakujemo naslednje uspešnosti:

- 50% uspešnost proti nizom z verjetnostno preferenco za kamen 20%. To so v resnici popolnoma naključni nizi.
- Proti nizom z verjetnostno preferenco za kamen 10% pričakujemo manjšo uspešnost, saj bodo ti nizi kamen igrali redkeje od ostalih, torej bo agent, ki vedno igra papir, pogosteje izgubljal.
- Proti vsem nizom, ki imajo verjetnostno preferenco za kamen višjo od 50%, pa je spodnja meja uspešnosti pravzaprav enaka verjetnosti preferenci. Na primer, če nizi igrajo kamen 70% časa, mi pa vedno igramo papir, bomo z gotovostjo zmagali vsaj 70% časa.

Zanimalo nas je torej, ali so vsi preučevani algoritmi res presegli spodnjo mejo uspešnosti, ki jo predstavlja agent, ki vedno igra papir. Iz tabele 4.11 je razvidno, da noben izmed preučevanih agentov ni bil slabši od spodnje meje uspešnosti, ampak je ta v večini bila presežena.

Verjetnost za kamen	MUPN	MV	SUPP	MKSU	MKIA
10%	51.00%	52.00%	53.00%	50.00%	53.00%
20%	50.00%	50.00%	50.00%	50.00%	50.00%
30%	51.00%	57.00%	53.00%	55.00%	54.00%
40%	54.00%	65.00%	59.00%	61.00%	61.00%
50%	58.00%	71.00%	66.00%	68.00%	68.00%
60%	65.00%	78.00%	72.00%	74.00%	75.00%
70%	72.00%	84.00%	79.00%	81.00%	81.00%
80%	81.00%	89.00%	85.00%	88.00%	87.00%
90%	90.00%	95.00%	90.00%	93.00%	93.00%

Tabela 4.11: Uspešnost algoritmov proti nizom z verjetnostno preferenco za kamen.

4.3 Dodatni eksperimenti

Zgornji eksperimenti so bili rutinsko izvedeni na vseh preučevanih algoritmih. Dodatno pa smo izvedli še dva eksperimenta, ki sta zahtevala prilagajanje programske kode dotičnih algoritmov.

4.3.1 Predprocesiranje

Eksperiment s predprocesiranjem je bil namenjen analizi občutljivosti markovskih verig. Pred napovedjo smo algoritmu predprocesirali (pokvarili) podatke, nato je le-ta poskušal napovedati na podlagi teh podatkov.

Postopek predprocesiranja je sledeč: na koraku, ko agent prejme napovedne podatke (v primeru markovskih verig je to vrstica iz matrike, ki vsebuje frekvence odigranih potez nasprotnika glede na stanje, v katerem se trenutno nahajamo), te podatke pretvorimo v verjetnosti. To storimo tako, da izračunamo vsoto vseh frekvenc, nato pa vsako frekvenco delimo z vsoto. Ko imamo podatke v tej obliki, lahko povečamo eno od verjetnosti, vse ostale pa ustrezno zmanjšamo. Odločili smo se za 10% povečanje verjetnosti za škarje. Dobljene podatke nato vrnemo algoritmu, ki jih obravnava enako kot sicer.

Pričakovani rezultati tega eksperimenta so nižje uspešnosti v vseh tekmah v primerjavi z markovskimi verigami brez predprocesiranja.

Pregled rezultatov

Predprocesirane markovske verige proti algoritmom. Markovske verige so se v boju proti ostalim algoritmom izkazale za stabilen algoritem, saj je predprocesiranje zelo malo vplivalo na njihovo uspešnost - opazili smo padec uspešnosti le za 0.78%.

	MK brez predprocesiranja	**MK s predprocesiranjem**
VP	60.00%	56.00%
VPS	86.00%	87.00%
VO	100.00%	100.00%
VK	100.00%	100.00%
MUPN	50.00%	48.00%
MK	50.00%	50.00%
Naključje	50.00%	50.00%
SUPP	50.00%	49.00%
MKSU	49.00%	48.00%
POVPREČNA USPEŠNOST	66.11%	65.33%

Tabela 4.12: Primerjava uspešnosti markovskih verig s predprocesiranjem in brez predprocesiranja v tekmi proti algoritmom.

Dejstvo, da so markovske verige spet dosegle 100% uspešnost proti algoritmu vedno kamen, vidno iz rezultatov iz tabele 4.12, je na prvi pogled morda presenetljivo. Kljub temu, da pred vsako potezo pokvarimo rezultate in dodamo škarjam 10% večjo možnost, da jih algoritem izbere kot najbolj verjetno potezo, jih ne bo v resnici nikoli izbral, saj iz matrike razbere 90% možnost za kamen in 10% možnost za škarje, torej vselej odigra papir.

Predprocesirane markovske verige proti nizom s fiksno preferenco. Proti nizom s fiksno preferenco smo opazili razliko le pri nizih s fiksno preferenco za kamen na 10 potez - pri vseh ostalih je ostala uspešnost markovskih verig s predprocesiranjem enaka povprečni uspešnosti izvirnega algoritma. Razlike so vidne v tabeli 4.13.

Povprečna uspešnost predprocesiranih markovskih verig je v tem tekmovanju znašala za 1.5% manj od uspešnosti izvirnega algoritma. markovske verige so se tako spet izkazale za dokaj stabilne.

Red	MK brez predprocesiranja	MK s predprocesiranjem
1	56.00%	53.00%
2	56.00%	54.00%
3	56.00%	55.00%
4	56.00%	54.00%
5	55.00%	54.00%
6	53.00%	53.00%
POVPREČNA USPEŠNOST	55.33%	53.83%

Tabela 4.13: Primerjava uspešnosti markovskih verig s predprocesiranjem in brez predprocesiranja v tekmi proti nizom s fiksno preferenco za kamen na 10 potez.

Predprocesirane markovske verige proti nizom z verjetnostno preferenco. Povprečna uspešnost predprocesiranih markovskih verig je v tem eksperimentu bila nižja za 0.67% v primerjavi z izvirnim algoritmom, iz tabele 4.14 pa je razvidno, da se ta razlika pojavi le na račun igranja proti nizom z verjetnostno preferenco 10% za kamen.

Verjetnost za kamen	MK brez predprocesiranja	MK s predprocesiranjem
10%	52.00%	47.00%
20%	50.00%	50.00%
30%	57.00%	57.00%
40%	65.00%	65.00%
50%	71.00%	71.00%
60%	78.00%	78.00%
70%	84.00%	84.00%
80%	89.00%	89.00%
90%	95.00%	95.00%
POVPREČNA USPEŠNOST	71.22%	70.56%

Tabela 4.14: Primerjava uspešnosti markovskih verig s predprocesiranjem in brez predprocesiranja v tekmi proti nizom z verjetnostno preferenco 10% za kamen

Ta rezultat nas je begal, zato smo ga večkrat ponovili, da smo se prepričali, da ne gre za zaključje. S podrobno analizo delovanja algoritma pa smo našli vzrok za to odstopanje.

Nizi z verjetnostno preferenco 10% za kamen igrajo kamen redkeje od ostalih potez; kamen igrajo namreč le 10% časa. Matrika v agentu z markovskimi verigami odraža to preferenco - kot lahko vidimo na sliki 4.1, je kamen po kateremkoli danem zaporedju odigran manj pogosto od ostalih potez, ostale poteze pa so relativno izenačene (število v prvem stolpcu pomeni,

kolikokrat je bil do sedaj igran kamen po zaporedju, ki ga predstavlja ključ vrstice). Ker mi predprocesiramo v smer škarij, bo algoritem pogosto sklepal, da bo nasprotnikova naslednja poteza najverjetneje škarje. Na podlagi tega bo odigral potezo, ki premaga škarje: kamen ali Spock.

```
rrr: 101; 221; 237; 242; 209;
rrp: 216; 512; 501; 524; 535;
rrs: 236; 504; 526; 552; 502;
rrL: 223; 518; 466; 540; 491;
rrS: 229; 518; 535; 515; 515;
rpr: 214; 550; 519; 518; 490;
```

Slika 4.1: Primerjava uspešnosti markovskih verig s predprocesiranjem in brez predprocesiranja v tekmi proti nizom s fiksno preferenco za kamen na 10 potez.

Slabša uspešnost v povprečju se pojavi zaradi primerov, ko algoritem izbere potezo Spock (in izbere jo pogosteje, kot ostale poteze). Nasprotnik - v tem primeru nizi z verjetnostno preferenco 10% za kamen - pogosto igra papir, škarje, kuščarja ali Spocka, redkeje igra kamen. Naša poteza - Spock - pa izgubi proti kuščarju in papirju, izenači proti Spocku, premaga pa škarje in redkeje igrani kamen. Zaradi naše preference za Spocka in nasprotnikove nizke preference za kamen, je Spock zelo slaba izbira. To je seveda posledica našega predprocesiranja, kar je tudi vzrok za 5% padec uspešnosti proti tem nizom.

Kot dodatni preizkus stabilnosti smo proti nizom z verjetnostno preferenco 10% za kamen igrali še z algoritmom, ki vedno igra Spocka - najslabši možen scenarij. Uspešnost tega algoritma je bila okoli 42%, kar je 5% manj od predprocesiranih markovskih verig. To lahko interpretiramo na naslednji način: struktura markovskih verig je stabilna do te mere, da nas je kljub predprocesiranju obvarovala pred najnižjo možno uspešnostjo in nam prihranila 5% pri razmerju zmag in porazov.

4.3.2 Optimalni meta-klasifikator

Zanimalo nas je, kolikšna je zgornja meja uspešnosti meta-klasifikatorja. Zgornjo mejo uspešnosti predstavlja meta-klasifikator, ki bi vedno znal izbrati tisti algoritem, ki je zmagal trenutno igro.

Optimalni meta-klasifikator smo simulirali z branjem nasprotnikove poteze pred izbiro algoritma. Potem smo le preverili, ali bi vsaj kateri izmed pomožnih algoritmov premagal to potezo. Če ja, smo to šteli za zmago, saj bi optimalni meta-klasifikator vedno znal izbrati zmagovalnega izmed algoritmov.

Pregled rezultatov

Rezultati proti naključnim nizom so pokazali, da je ta eksperiment brez pravega smisla - dosegli smo namreč uspešnost nad 90%. Rezultati proti vsem ostalim testnim scenarijem (ostali algoritmi, preferenčni nizi) so bili podobni. Tako visoko uspešnost je bilo moč doseči zaradi dejstva, da se iz množice pomožnih algoritmov, iz katere meta-klasifikator izbira, zelo pogosto najde vsaj eden, ki tako ali drugače odigra potezo, ki premaga nasprotnikovo. Optimalni meta-klasifikator pa seveda ve, katerega mora izbrati, tudi, če nasprotnik igra naključno.

Prav tako bi se visoka uspešnost pojavila v primeru, da bi za pomožne algoritme vzeli več instanc agenta, ki igra naključno. Optimalni meta-klasifikator bi vselej izbral enega izmed tistih, ki so po naključju premagali nasprotnikovo, prav tako naključno, potezo. Naš eksperiment je bil torej slabo zastavljen, saj odgovor na zastavljeno vprašanje ne pove ničesar o samem napovedovanju in učenju iz vzorcev.

Poglavje 5

Zaključek

V tem delu smo z različnimi algoritmi igrali igro Kamen, papir, škarje, kuščar, Spock in na podlagi zgodovine odigranih potez poskušali najti vzorce v nasprotnikovi igri.

Zastavili smo si tri različne testne scenarije, s katerimi smo preizkušali implementirane algoritme: igra proti drugim algoritmom, igra proti nizom s fiksno preferenco za kamen in igra proti nizom z verjetnostno preferenco za kamen. Ugotovili smo, da se je v boju med algoritmi najbolj izkazal meta-klasifikator z izbranimi algoritmi, algoritem z markovskimi verigami pa je prevladal pri igrah proti obem vrstam nizov.

Dodatno smo izvedli še dva eksperimenta. Prvi je bil eksperiment s predprocesiranjem markovskih verig, s katerim smo testirali njihovo stabilnost. Izkazalo se je, da so markovske verige kljub 10% šumu podatkov v stran škarij ohranile skoraj enako uspešnost kot izvirni algoritem. Drugi eksperiment je bil namenjen ocenjevanju zgornje meje izboljšave meta-klasifikatorja. Za zgornjo mejo smo vzeli optimalni meta-klasifikator, ki smo ga simulirali z branjem nasprotnikove poteze preden smo odigrali svojo potezo. Pričakovali smo zgovorno oceno o zgornji meji, vendar se je izkazalo, da tak meta-klasifikator prevlada tudi nad naključnimi nizi z uspešnostjo preko 90%. Taka uspešnost je nerealna in ni odsev uspešnosti posameznih pomožnih algoritmov, med katerimi meta-klasifikator izbira, ampak se pojavi kot stranski učinek dej-

stva, da je možnih potez le pet. Ker je število potez tako omejeno, se med pomožnimi algoritmi skoraj vedno najde vsaj eden, ki je na dano nasprotnikovo potezo odigral zmagovalno potezo.

Za nadaljnje raziskovanje na tem področju bi bil smiseln temeljit pregled različnih vrednosti parametrov Q-učenja in vpliv le-teh na uspešnost algoritmov, ki temeljijo na spodbujevanem učenju na podlagi Q-učenja. Na tak način bi lahko določili optimalne parametre za učenje vzorcev pri tej igri. Testiranje markovskih verig s predprocesiranjem bi razširili z različno vrednostjo predprocesiranja v smer ene od potez, dodatno bi lahko dodali predprocesiranje v smer več potez hkrati.

Literatura

[1] Darse Billings. The first international RoShamBo programming competition. *ICGA Journal*, 23(1):42–50, 2000.

[2] Raymond Hettinger. Pattern Recognition and Reinforcement Learning. Dosegljivo: https://rhettinger.github.io/rock_paper.html, 2019. [Dostopano 1. 12. 2019].

[3] Ishikawa Senoo Laboratory. Janken (rock-paper-scissors) robot with 100% winning rate (human-machine cooperation system). Dosegljivo: https://ishikawa-vision.org/fusion/Janken/index-e.html, 2012. [Dostopano 1. 12. 2019].

[4] Daniel Lawrence Lu. Rock paper scissors algorithms. Dosegljivo: https://daniel.lawrence.lu/programming/rps/, 2019. [Dostopano 1. 12. 2019].

[5] Gabriele Pozzato, Stefano Michieletto, and Emanuele Menegatti. Towards Smart Robots: Rock-Paper-Scissors Gaming versus Human Players. In *PAI@ AI* IA*, pages 89–95. Citeseer, 2013.

[6] Douglas Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, pages 827–832, 2015.

[7] RPSContest. Rock Paper Scissors Programming Competition. Dosegljivo: http://www.rpscontest.com/, 2011. [Dostopano 17. 12. 2019].

[8] Devin Soni. Introduction to Markov Chains. Dosegljivo: https://www.

kdnuggets.com/2018/03/introduction-markov-chains.html, 2018. [Dostopano 1. 12. 2019].

[9] Techopedia Staff. Q-learning. Dosegljivo: https://www.techopedia.com/definition/32882/q-learning, 2019. [Dostopano 4. 12. 2019].

[10] Techopedia Staff. Reinforcement Learning. Dosegljivo: https://www.techopedia.com/definition/32055/reinforcement-learning, 2019. [Dostopano 4. 12. 2019].

[11] James Stanley. A Rock-Paper-Scissors AI that is too good. Dosegljivo: https://incoherency.co.uk/blog/stories/rock-paper-scissors.html, 2018. [Dostopano 1. 12. 2019].

[12] Uptownscience. How To Win At Rock-Paper-Scissors. Dosegljivo: https://uptownscience.files.wordpress.com/2015/08/rock-paper-scissors-lizard-spock.png, 2015. [Dostopano 15. 01. 2020].

[13] Sony E Valdez, Generino P Siddayao, and Proceso L Fernandez. The Effectiveness of Using a Modified "Beat Frequent Pick" Algorithm in the First International RoShamBo Tournament. *International Journal of Information and Education Technology*, 5(10):740–747, 2015.

[14] Wikipedia. Rock paper scissors. Dosegljivo: https://upload.wikimedia.org/wikipedia/commons/thumb/6/67/Rock-paper-scissors.svg/1200px-Rock-paper-scissors.svg.png, 2020. [Dostopano 15. 01. 2020].

[15] The world rock paper scissors association. Rock Paper Scissors Advanced Strategies. Dosegljivo: https://www.wrpsa.com/rock-paper-scissors-advanced-strategies/, 2019. [Dostopano 16. 12. 2019].

[16] The world rock paper scissors association. The official rules of rock paper scissors. Dosegljivo: https://www.wrpsa.com/the-official-rules-of-rock-paper-scissors/, 2019. [Dostopano 16. 12. 2019].