

UNIVERZA V LJUBLJANI
FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Andrej Babič

Drevesno preiskovanje po metodi Monte Carlo pri igri poker

DIPLOMSKO DELO NA UNIVERZITETNEM ŠTUDIJU

MENTOR: izr. prof. dr. Branko Šter

Ljubljana, 2013



Št. naloge: 01886/2012

Datum: 04.12.2012

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **ANDREJ BABIČ**

Naslov: **DREVESNO PREISKOVANJE PO METODI MONTE CARLO PRI IGRI
POKER**

MONTE CARLO TREE SEARCH IN POKER GAMES

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

Implementirajte metodo drevesnega preiskovanja Monte Carlo in jo uporabite na primeru igranja pokra, varianta Texas Hold'em. Raziščite vpliv parametrov metode drevesnega preiskovanja na uspešnost igranja. Kot nasprotnike uporabite različne tipe agentov. Ocenite, ali je omenjena metoda primerna za računalniško igranje pokra.

Mentor:

izr. prof. dr. Branko Šter

Dekan:

prof. dr. Nikolaj Zimic



Izjava o avtorstvu diplomskega dela

Spodaj podpisani Andrej Babič, z vpisno številko **63050005**, sem avtor diplomskega dela z naslovom:

Drevesno preiskovanje po metodi Monte Carlo pri igri poker

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Branka Štera,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki »Dela FRI«.

V Ljubljani, dne 14.6.2013

Podpis avtorja:

Zahvala

Rad bi se zahvalil mentorju, izr. prof. dr. Branku Šteru, za izkazano strokovno podporo in številne popravke med pisanjem diplomskega dela. Posebno cenim vsestransko pomoč in nasvete prijatelja Toma, zahvala pa velja tudi vsem sodelavcem in prijateljem, ki so mi s tehničnim znanjem pomagali do uspešnega zaključka diplomskega dela.

Posebna zahvala gre staršema, ki sta mi študij omogočila in me pri njem spodbujala.

Hvala tudi puncu Katarini za nenehno podporo in pomoč, ter svetovanje in kritično analizo tekom celotnega študija.

Kazalo

Povzetek	1
Abstract	2
1 Uvod	3
1.1 Cilji diplomske naloge	3
2 Teoretična osnova	4
2.1 Strojno učenje	4
2.1.1 Spodbujevalno učenje	5
2.1.2 Metode Monte Carlo	6
2.1.3 Preiskovalno drevo	7
2.1.4 Preiskovalno drevo Monte Carlo	8
2.2 Texas Hold'em	13
2.2.1 Opis igre	13
2.2.2 Vrednotenje rok	15
2.2.3 Stave	16
2.2.4 Odločanje	17
2.3 MCTS za Texas Hold'em	18
2.3.1 Sestop	18
2.3.2 Širitev	19
2.3.3 Simulacija	19
2.3.4 Vzratni prenos	20
2.4 Parametri simulacije	20
2.4.1 Število iteracij algoritma MCTS	20
2.4.2 Število simulacij na iteracijo algoritma MCTS	20
2.4.3 Strategija izbiranja vozlišč	21
2.4.4 Model nasprotnika	21
2.4.5 Ohranjanje drevesa med simulacijami	21
3 Orodja	22
3.1 Poker Academy	22
3.2 Meerkat API	23
3.2.1 Primer igre	24
3.3 OpenTestBed	25

3.4	Obstoječi agenti	26
3.4.1	AlwaysCallBot	27
3.4.2	FlockBot.....	27
3.4.3	SimpleBot.....	27
3.4.4	ChumpBot	28
4	Izvedba	29
4.1	Ideja o izvedbi	29
4.1.1	Izdelava drevesnega preiskovanja Monte Carlo.....	29
4.1.2	Izdelava agenta za poker Texas Hold'em.....	29
4.1.3	Izdelava načinov za zajemanje in prikazovanje podatkov	30
4.1.4	Izdelava načrta izvajanja poskusov	30
4.1.5	Priprava primerjalnih podatkov	31
4.2	Rezultati izvedbe	32
4.2.1	Ogrodje MCTS	32
4.2.2	Poker agent za »Meerkat API«.....	33
4.2.3	Pridobljeni podatki.....	33
4.2.4	Težave	34
5	Rezultati	35
5.1	Analiza vpliva parametrov algoritma MCTS	36
5.1.1	Strategija izbiranja vozlišč	36
5.1.2	Ohranjanje drevesa	38
5.1.3	Model nasprotnika	40
5.1.4	Število simulacij na iteracijo algoritma MCTS.....	41
5.2	Analiza igre proti različnim agentom	42
5.2.1	AlwaysCallBot	42
5.2.2	FlockBot.....	44
5.2.3	SimpleBot.....	45
5.2.4	ChumpBot	46
5.3	Primerjava rezultatov igre proti različnim agentom	47
6	Zaključek.....	48
7	Stvarno kazalo	49
8	Viri in literatura	51

Povzetek

Drevesno preiskovanje Monte Carlo postaja v zadnjih letih vedno bolj uporabljana metoda umetne inteligence. Zanimanje zanjo vzbuja predvsem relativno enostavna implementacija in zelo malo (praktično nič) potrebnega znanja o domeni, ki jo želimo modelirati.

V diplomskem delu bom predstavil implementacijo in analizo vpliva različnih parametrov drevesnega preiskovanja Monte Carlo na igro poker Texas Hold'em. Glavni izziv je pri delu predstavljala izdelava agenta, ki bo brez specifičnega znanja o pokru Texas Hold'em igral zadovoljivo dobro, pri tem pa za posamezno odločitev ne bo porabil preveč časa. Glavna težava drevesnega preiskovanja Monte Carlo je namreč ravno njegova počasnost v primerjavi z drugimi metodami, ki uporabljajo več specifično domenskega znanja.

Na spletu je na voljo kar nekaj ogrodij za razvoj poker agentov, med katerimi je eden najbolj znanih »Meerkat API«. Ogrodje ima veliko omrežno skupnost, zato je dostopnih mnogo primerov algoritmov za igranje pokra, ki so osnovani na njem. Ker sem tudi sam za razvoj svojega agenta uporabil »Meerkat API«, nisem imel težav z iskanjem nasprotnikov za primerjavo.

V sklopu diplomskega dela je nastalo tudi splošno ogrodje za implementacijo drevesnega preiskovanja Monte Carlo. Ogrodje je napisano v Javi, je odprtokodno in prosto dostopno na spletu. Omogoča poljubno prilagajanje algoritma specifičnim potrebam in spremljanje simulacije z raznimi prikazovalniki.

Rezultati simulacij so pokazali, da je drevesno preiskovanje Monte Carlo lahko zelo dober nasprotnik v igri Texas Hold'em. Na žalost pa je njegova hitrost odločanja manjša kot pri običajnih, domensko specifičnih algoritmihih. K njegovi počasnosti pripomore tudi dejstvo, da je napisan v Javi.

Ključne besede: umetna inteligenca, Monte Carlo, drevesno preiskovanje, poker, Texas Hold'em, Meerkat API, OpenTestBed

Abstract

As a method of artificial intelligence, Monte Carlo tree search has been widely adopted in the last few years. The main reasons for this are its relatively simple implementation and a very low domain specific knowledge needed for it to work properly.

In my thesis, I will present its implementation and analyse the key parameters for the Texas Hold'em poker game. The main challenge was to develop an agent which would be able to play poker on an acceptable level without consuming too much time for each decision. One of the main problems of the Monte Carlo tree search, compared with other methods that rely more heavily on domain specific knowledge, is its relative slowness.

There are a lot of poker frameworks available online. One of the most popular is Meerkat API. Its online community is large, thus there are a lot of poker playing algorithms using it. Since my agent has been developed with Meerkat API as well, I have had no problems finding suitable opponents for the simulations.

Within this thesis, a generic Java framework for Monte Carlo Tree Search algorithms was developed. The framework is open source and freely available online. It enables us to customize each portion of the Monte Carlo Tree Search algorithm, as well as gives us the possibility to monitor our simulations with several data visualizers.

The simulation results have shown that a Monte Carlo Tree Search simulation can be a very challenging opponent in a game of Texas Hold'em. Unfortunately, its decision-making speed cannot be compared to other algorithms with more domain specific knowledge. Its speed is also further impaired by the fact that it is written in Java.

Keywords: artificial intelligence, Monte Carlo, tree search, poker, Texas Hold'em, Meerkat API, OpenTestBed

1 Uvod

1.1 Cilji diplomske naloge

V sklopu diplomske naloge bom analiziral algoritem drevesnega preiskovanja Monte Carlo (v nadaljevanju MCTS) na primeru igre z nepopolnimi podatki. Implementiral bom agent, ki bo igral poker igro Texas Hold'em. Igro sem izbral, ker ima dovolj enostavna pravila, rezultate iger pa je lahko spremljati in objektivno ovrednotiti.

Nasprotniki bodo agenti, ki uporabljajo različne deterministične ali psevdo deterministične algoritme. Ker imajo različni agenti različne lastnosti igranja, (nekateri so agresivni, drugi pasivni, nekateri blefirajo itd.), bom svojega agenta preveril na različnih elementih igre in s tem poskušal poiskati optimalno sestavo parametrov agenta.

S simulacijo iger bom poskušal ugotoviti:

1. vpliv različnih parametrov algoritma MCTS na njegovo uspešnost,
2. uspešnost algoritma MCTS v primerjavi z različnimi psevdo determinističnimi agenti,
3. in ali je algoritem MCTS primerna izbira za igro poker Texas Hold'em.

2 Teoretična osnova

Za lažje sledenje eksperimentom in boljše razumevanje njihovih zaključkov je potrebno dobro teoretično poznavanje uporabljenih metod in domene, nad katero eksperimentiramo. V tem poglavju bom predstavil osnovne koncepte, ki so sestavni del algoritma za drevesno preiskovanje Monte Carlo (v nadaljevanju MCTS). Opisal bom tudi izbrano igro, poker Texas Hold'em, in določil metrike, s katerimi bom vrednotil rezultate poskusov.

2.1 Strojno učenje

Cilj strojnega učenja je poiskati pravila in povezave v določeni množici podatkov tako, da lahko te informacije kasneje uporabimo za napovedovanje čim bolj optimalnega ravnanja v neznani situaciji.

Če se osredotočimo na temo diplomske naloge, lahko za primer podamo agent, ki mu določimo osnovna pravila igre poker in osnovna pravila, na podlagi katerih naj se v dani situaciji odloči. Agent pustimo igrati dlje časa. S časom naj bi igra agenta postajala vedno boljša, oziroma naj bi se pravila, na podlagi katerih se odloča, vedno bolj približevala optimalnim; v našem primeru to pomeni, da bi agent imel vedno večji dobiček.

Strojno učenje lahko delimo na podlagi vrste nadzora in adaptivnosti [16].

Vrste strojnega učenja glede na nadzor:

- Nadzorovano učenje: Poteka na podlagi parov, ki so sestavljeni iz vhodnih spremenljivk in želenih rezultatov. Določena metoda na podlagi parov izdelava algoritem, ki naj bi znal pravilno predvidevati rezultat tudi na do tistega trenutka še ne znanem vhodu.
- Nenadzorovano učenje: Poteka samo na podlagi vhodnih vrednosti. Sistem skuša ugotoviti vzorce v podatkih. Od nadzorovanega učenja se razlikuje v tem, da pravih izhodov ne poznamo.
- Spodbujevalno učenje: Osnova je vrednotenje rezultata z nagrado ali s kaznijo. Glede na vhodne podatke in trenutno stanje se algoritem odloča za akcije, ki so mu v preteklosti prinesle največ nagrade.

Vrste strojnega učenja glede na adaptivnost:

- »Offline« učenje: Celotno učenje se izvede pred uporabo stroja. Algoritem se med samim delovanjem stroja ne izpopolnjuje.

- Inkrementalno učenje: Uporabimo »offline« učenje. Stroj uporabljamo, dokler ne zaznamo težav pri delovanju algoritma. Stroj ponovno učimo »offline« in ga izpopolnjujemo. Stroj uporabljamo naprej.
- Sprotno ali »online« učenje: Stroj se med delovanjem nenehno izpopolnjuje in uči na novih primerih. Učenje je primerno za kompleksno in spreminjajoče se okolje, ker se mu stroj lahko sproti prilagaja.

Algoritem MCTS lahko klasificiramo kot sprotno spodbujevalno učenje.

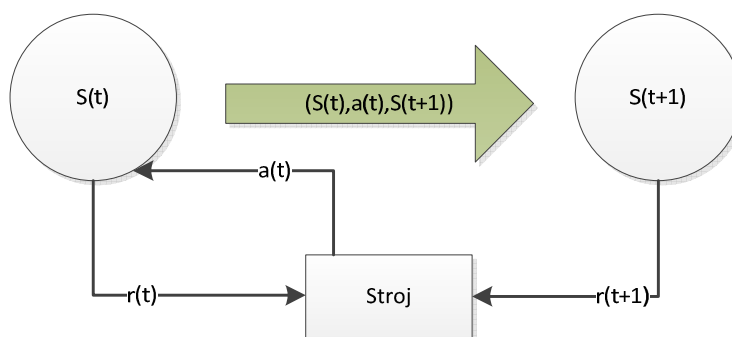
2.1.1 Spodbujevalno učenje

Spodbujevalno učenje skuša odgovoriti na vprašanje, katero akcijo je potrebno v dani situaciji izvesti, da bi kot rezultat akcije v prihodnosti dobili čim večjo nagrado [15].

Osnovni model spodbujevalnega učenja [7] je sestavljen iz:

- množice stanj okolice – S ,
- množice akcij stroja – a ,
- pravil prehodov med stanji okolice – (S_t, a_t, S_{t+1}) ,
- pravil za določanje nagrade – r ,
- in pravil za opazovanje okolice – informacije, ki jih lahko stroj prejme od okolja.

Z optimalno izbiro akcije a skuša stroj v danem stanju okolja S na dolgi rok maksimizirati prejeto nagrado r .



Slika 1: Interakcija stroja z okoljem.

Stroj za interakcijo z okoljem čas diskretizira. V času t je okolica v stanju S_t . Stroj od okolice prejme nagrado r_t . Iz množice akcij stroj izbere akcijo a_t , katera povzroči spremembo stanja okolice v S_{t+1} in posledično tudi prejetje nagrade r_{t+1} . Prehod stanja okolja na podlagi akcije je enolično določen s prehodom (S_t, a_t, S_{t+1}) (Slika 1).

Glede na dinamično programiranje je spodbujevalno učenje primerno za okolja z velikim številom stanj ali zveznim prostorom, oziroma v primerih, ko nimamo vseh podatkov o okolju. Uspešno se uporablja v primerih ko:

- imamo na razpolago samo simulacijo okolja, modela pa ne,
- o okolju ne vemo ničesar in je edini način za pridobitev informacij o okolju interakcija z njim.

Ena ključnih lastnosti spodbujevalnega učenja je raziskovanje novih akcij. Pojavi se problem, kako in kdaj izbrati do zdaj še nepreizkušeno akcijo. V kolikor želimo algoritem skozi čas izboljšati, mora le-ta poskušati nove akcije, da bi lahko ugotovil, katera vodi do skupno največje nagrade. Naključno izbiranje akcij daje zelo slabe rezultate. Ravnoesje med izkoriščanjem že znanih akcij in raziskovanjem novih je ključnega pomena za uspešnost algoritma. Ena najbolj enostavnih in običajno najbolj uporabljenih metod je tako imenovana metoda ε – *greedy* (ε – *požrešna*). Pri tej metodi stroj izbere akcijo, za katero verjame, da mu bo dolgoročno prinesla najvišjo nagrado z verjetnostjo $1 - \varepsilon$, medtem ko z verjetnostjo ε izbere naključno akcijo. S spreminjanjem parametra $0 < \varepsilon < 1$ spreminjamo stopnjo raziskovanja novih akcij. Med delovanjem stroja lahko parameter ε dinamično spreminjamo, da na primer preizkuša manj novih akcij, ko je njegovo delovanje skoraj optimalno.

Za nadzor spodbujevalnega učenja obstajajo številni algoritmi. V diplomski nalogi se bom osredotočil na metode Monte Carlo.

2.1.2 Metode Monte Carlo

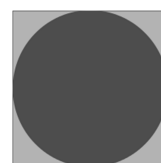
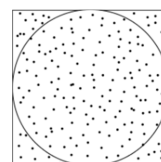
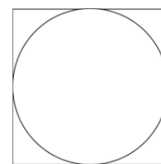
Metode Monte Carlo se uporabljajo v primerih, ko je nemogoče ali preveč zahtevno izračunati rezultat sistema z determinističnim algoritmom – na primer pri problemu medsebojnega vpliva gravitacijskega privlaka velikega števila teles (ang. many-body problem) [8].

Osnovni pristop vseh metod Monte Carlo lahko na grobo razdelimo na štiri korake:

1. Definicija domene možnih vhodnih spremenljivk.
2. Naključno generiranje vhodnih spremenljivk iz domene na podlagi verjetnostne porazdelitve.
3. Deterministični izračun rezultata na podlagi generiranih vhodnih spremenljivk.
4. Interpretacija rezultatov determinističnega izračuna.

Delovanje metod Monte Carlo se enostavno ponazori na problemu računanja razmerja med premerom in obsegom kroga – π . Za vsakega od zgoraj naštetih korakov bom opisal postopek na danem primeru:

1. Domena možnih vhodnih vrednosti je omejena s kvadratom, ki ima včrtan krog. Stranica kvadrata je enaka premeru kroga. Možne vhodne vrednosti so vse točke znotraj kvadrata.
2. Naključno generiramo točke v kvadratu. Več točk kot generiramo, bolj natančen bo naš končni izračun.
3. Preštejemo samo število točk, ki se nahajajo v včrtanem krogu; preštejemo skupno število vseh točk v kvadratu.
4. Razmerje med številom točk v krogu in skupnim številom točk je enako razmerju med površino kroga in površino kvadrata. (To bi bilo seveda res samo, če bi generirali neskončno število točk. Ker smo generirali končno število točk, gre v našem primeru zgolj za približek). Verjetnost, da se točka nahaja v krogu, je enaka odstotku površine kvadrata, ki jo zaseda krog.



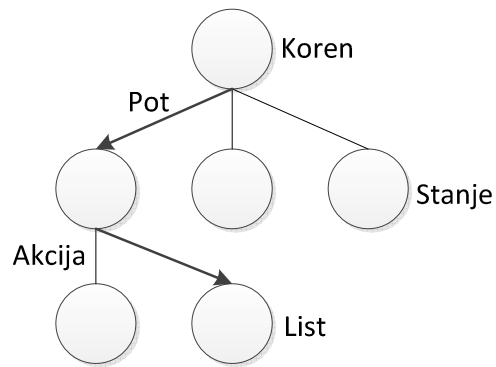
Iz enačbe za površino kroga $S_c = \pi(\frac{D}{2})^2$ in enačbe za površino kvadrata $S_{sq} = D^2$ izhaja, da lahko π izračunamo kot $\pi = 4 * \frac{S_c}{S_{sq}}$.

Razmerje med površino kroga S_c in površino kvadrata S_{sq} , katero smo dobili v 4. točki, je torej potrebno samo še pomnožiti s 4, da dobimo π .

Zaradi enostavnega pristopa pri reševanju kompleksnih problemov se v zadnjih letih metode Monte Carlo razširjajo v algoritmih za igranje kompleksnih iger. Uspešno so bile implementirane na igrah kot so tantriks, potapljanje ladjic, go in poker. Skupno vsem tem algoritmom je to, da uporabljajo drevesno preiskovanje Monte Carlo.

2.1.3 Preiskovalno drevo

Preiskovalno drevo je struktura, ki je definirana kot aciklični graf, kjer ima vsako vozlišče natanko eno nadrejeno vozlišče (ang. parent) in nič ali več podrejenih vozlišč (ang. child) (Slika 2). Izjema je koren (ang. root) drevesa, ki nima nadrejenega vozlišča [1,5]. Vozlišče, ki nima podrejenih vozlišč, poimenujemo list (ang. leaf). Pot (ang. path) v grafu definiramo kot zaporedje akcij, s katerimi prehajamo med določenimi vozlišči. Preiskovalno drevo uporabljamo pri problemih, ki jih je mogoče predstaviti v obliki stanj (vozlišč) in akcij (povezave med vozlišči).



Slika 2: Prikaz preiskovalnega drevesa.

Domene, ki jih rešujemo s preiskovalnimi drevesi, lahko razdelimo na stohastične in deterministične. Pri determinističnih problemih oziroma procesih verjetnost ne vpliva na prehod med stanji. Posledično z enakimi začetnimi parametri vedno pridemo do enakega končnega stanja. Stohastični sistemi se od determinističnih razlikujejo v tem, da vključujejo verjetnost. Vsaj en parameter sistema ni natančno določen, ampak ga običajno opisujemo z verjetnostno porazdelitvijo.

Pri preiskovalnih drevesih v kontekstu teorije iger lahko probleme (igre) še dodatno razdelimo glede na število igralcev:

- en igralec – gre za optimizacijske probleme, kjer ne tekmujemo z nasprotnikom (npr. sestavljanje),
- dva igralca – problemi, kjer igramo z nasprotnikom (npr. potapljanje ladjic),
- več igralcev – problemi, kjer igramo z več nasprotniki (npr. poker).

S preiskovalnim drevesom skušamo za dano stanje (igro) izbrati najboljšo akcijo (oziroma zaporedje akcij) tako, da zadovoljimo določen kriterij optimalnosti.

Poker je igra, ki jo modeliramo kot stohastični proces. Verjetnost – sreča – je namreč ključnega pomena pri igri. Čeprav poker uvrščamo v igro z več igralci, se bomo v sklopu tega besedila osredotočili na probleme z dvema igralcema, in sicer zaradi manjšega vpliva verjetnosti na izid igre (manjše število verjetnostnih parametrov), ter zaradi manjšega števila parametrov odločanja (npr. postavitve različnih igralcev za mizo).

2.1.4 Preiskovalno drevo Monte Carlo

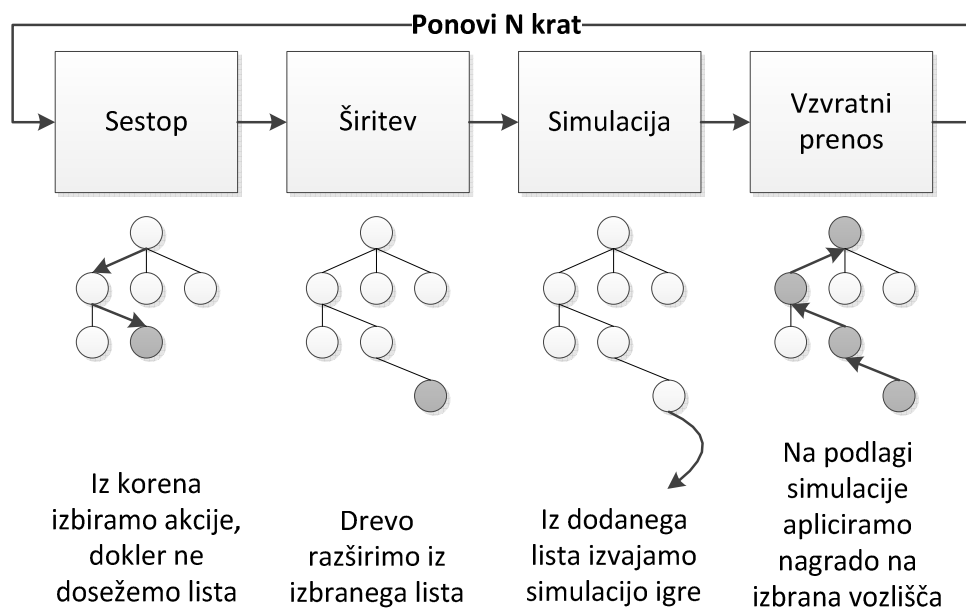
Drevesno preiskovanje Monte Carlo (ang. Monte Carlo Tree Search) je algoritem za drevesno preiskovanje, ki je osnovan na metodah Monte Carlo. Osnovno delovanje algoritma lahko opišemo kot simuliranje velikega števila iger od določenega stanja do konca igre [1,2,5]. Simulacije izvajamo, da ugotovimo, katera akcija nam z največjo verjetnostjo prinese želeni rezultat. Z upoštevanjem predhodnih simulacij se rezultati postopno izboljšujejo. Algoritem se osredotoči na tiste poti v preiskovalnem drevesu, ki so v preteklosti prinesle najboljše

rezultate. Ko je rezultat simulacije znan, se nagrada/kazen aplicira na obiskano pot v drevesu. Z večanjem števila odigranih iger postanejo poti, ki v preiskovalnem drevesu prinašajo največ nagrade, vedno bolj izrazite.

Vsako vozlišče v drevesu hrani vsaj dva parametra:

- števec, ki pove, kolikokrat je bilo vozlišče obiskano, in
- trenutno vrednost vozlišča.

V kolikor je bilo vozlišče v dani simulaciji obiskano, se obe vrednosti posodobita. Števec se vedno poveča za 1, medtem ko se trenutna vrednost vozlišča popravi glede na rezultat simulacije in algoritem za vzvratni prenos.



Slika 3: Prikaz drevesnega preiskovanja Monte Carlo.

Ena iteracija algoritma je sestavljena iz štirih faz (Slika 3):

1. Sestop: Akcije so izbrane na podlagi statistike iz predhodno igranih iger. Izbere se akcijo, za katero se verjame, da lahko prinese najvišji potencialni dobiček.
2. Širitev: Ko imamo izbrani list (vozlišče brez podrejenih vozlišč), drevo razširimo z enim ali več podrejenimi vozlišči.
3. Simulacija: Opravi se simulacija iz razširjenega lista drevesa. Igra se simulira na podlagi simulacijskih pravil. Cilj simulacije je ovrednotiti list, s katerim smo razširili drevo.
4. Vzratni prenos: Ko je rezultat simulacije znan, se posodobi vsako obiskano vozlišče glede na rezultat igre.

Za lažjo predstavo bomo algoritem zapisali tudi v obliki psevdo kode (Slika 4).

```

/* Algoritem izvajamo, dokler imamo še čas za odločanje */
while(hasTime)
{
    /****** 1. KORAK *****/
    /* Začetno vozlišče je koren drevesa */
    currentNode = rootNode;
    /* Preišči drevo, dokler ne prideš do lista */
    while (currentNode.hasChilds())
    {
        /* Izberi primerno podrejeno vozlišče - v kolikor obstaja */
        currentNode = SelectChild(currentNode);
    }

    /****** 2. KORAK *****/
    /* Zadnji obiskani list je trenutni list */
    lastNode = currentNode;
    /* Listu dodaj novo podrejeno vozlišče */
    currentNode = Expand(lastNode);

    /****** 3. KORAK *****/
    /* Simuliraj igro iz zadnjega dodanega vozlišča */
    gameResult = SimulateGame(currentNode);

    /****** 4. KORAK *****/
    /* Preišči vsa vozlišča, ki so bila obiskana v 1. koraku */
    while (currentNode.visited())
    {
        /* Apliciraj nagrado */
        BackPropagate(currentNode, gameResult);
        /* Izberi nadrejeno vozlišče */
        currentNode = currentNode.parent();
    }
}
/* Vrni najboljšo akcijo */
return BestAction(rootNode);

```

Slika 4: Psevdo koda algoritma drevesnega preiskovanja Monte Carlo.

Značilnost drevesnega preiskovanja Monte Carlo je, da se z večanjem števila simulacij večja natančnost odločanja in s tem možnost za odločitev, ki bo prinesla želeni rezultat. Število simulacij je običajno omejeno s časom, ki ga imamo na razpolago za odločitev.

Drevo začnemo preiskovati v izhodišču. V vsakem vozlišču se odločimo, katero podrejeno vozlišče bomo obiskali. Postopek nadaljujemo, dokler ne pridemo do vozlišča brez podrejenih vozlišč – list drevesa. Vozlišča izbiramo s funkcijo **SelectChild**, ki se odloča na podlagi dosedanjih parametrov v trenutnem vozlišču.

Iz izbranega lista drevo razširimo s funkcijo **Expand**. Funkcija lahko vsebuje deterministična pravila za določanje novega lista drevesa.

Ko smo drevo ustrezno razširili, iz na novo dodanega lista poženemo simulacijo. Simulacija upošteva simulacijska pravila. Običajno so to pravila igre in ostale značilnosti, ki določajo naš

agent. Simulacijo opravimo s funkcijo **SimulateGame**. Rezultat simulacije je izid simulirane igre iz na novo dodanega lista.

Na podlagi rezultata simulacije igre posodobimo vsa ali le del vozlišč, ki so bila udeležena pri izidu simulacije. To storimo s klicem funkcije **BackPropagate**. Funkcija ovrednoti rezultat igre za posamezno vozlišče in tako spremeni njegovo vrednost.

Iz opravljenih simulacij dobimo popravljeno preiskovalno drevo. S funkcijo **BestAction** izberemo najboljšo akcijo za trenutno situacijo.

Osnovne akcije v psevdo kodi (SelectChild, Expand, SimulateGame, BackPropagate) predstavljajo osnovne faze v drevesnem preiskovanju Monte Carlo. Ker so te metode ključnega pomena za razumevanje algoritma, jih bomo v nadaljevanju podrobneje predstavili.

2.1.4.1 Sestop

Šestop je rekurzivna funkcija, ki preišče drevo od izhodišča do lista tako, da izbira (predvidoma) čim bolj ustrezne akcije.

Strategija izbiranja vozlišč (ang. selection policy) uravnava ravnovesje med izkoriščanjem že znanih akcij in raziskovanjem novih. Raziskovanje novih akcij je potrebno zaradi nepopolnega poznavanja domene. Dejstvo, da določena akcija ni bila raziskana, ne pomeni, da ne prinaša želenega rezultata, oziroma da ni dobičkonosna.

Običajno je pri strategiji izbiranja vozlišč pomembno tudi, kolikokrat je bilo določeno vozlišče obiskano. Večkrat kot smo izvajali simulacijo za določeno vozlišče, bolj zanesljiva je njegova vrednost.

Za izbiranje vozlišč obstaja veliko algoritmov, vendar se bom omejil na dva:

- naključni (ang. random),
- UCT (ang. Upper Confidence Bounds for Trees).

Naključna izbira akcije

Akcije izbiramo naključno z enakomerno porazdelitvijo verjetnosti med vsemi možnimi akcijami. Tak način izbiranja ne zgolj omogoča optimalno delovanje, ampak zagotavlja tudi največjo možno mero raziskovanja drevesa.

Upper Confidence Bounds for Trees (Algoritem zgornje meje zaupanja drevesu)

Metoda je enostavna za implementacijo in je bila že večkrat uspešno uporabljena pri različnih algoritmihih MCTS [6] (npr. pri MCTS za igro go). Algoritem upošteva tako število obiskov kot pričakovano vrednost vozlišč. V algoritmu nastopajo spremenljivke:

- izhodiščno vozlišče p ,
- število obiskov izhodiščnega vozlišča n_p ,
- množica podrejenih vozlišč I , ki so dosegljiva iz trenutnega vozlišča,

- trenutno izbrano podrejeno vozlišče i ,
- pričakovana vrednost trenutno izbranega podrejenega vozlišča v_i ,
- število obiskov trenutno izbranega podrejenega vozlišča n_i ,
- koeficient C , ki predstavlja eksperimentalno optimizirano konstanto.

UCT računamo po enačbi:

$$k \in \operatorname{argmax}_{i \in I} \left(v_i + C * \sqrt{\frac{\ln n_p}{n_i}} \right)$$

Izmed množice podrejenih vozlišč I , ki imajo skupno nadrejeno vozlišče p , izberemo vozlišče k , katerega vrednost je po podani enačbi največja.

2.1.4.2 Širitev

Funkcija drevo razširi z novim listom ali listi. Posledično se število listov veča s številom iteracij, kar pomeni, da imamo z večanjem števila iteracij bolj natančno preiskovalno drevo.

Običajno v vsaki iteraciji dodamo en list. Obstajajo tudi druge, bolj zapletene strategije, vendar pa ima strategija za dodajanje listov drevesu navadno manjši vpliv na končni uspeh algoritma, zato običajno bolj kompleksnih strategij ne uporabljamo.

Na začetku, dokler izmed nabora možnih akcij, oziroma prehodov iz izbranega stanja, še nismo izbrali vseh akcij vsaj enkrat, med še ne izbranimi akcijami izbiramo naključno.

2.1.4.3 Simulacija

Simuliranje se izvede iz na novo dodanega lista drevesa. Izmed vseh možnih akcij v določenem stanju izberemo naslednjo akcijo tako, da zadostimo simulacijski strategiji (ang. default policy).

Najbolj enostavna simulacijska strategija je naključno izbiranje med možnimi akcijami. Ker je izid simulacije ključnega pomena za aplikacijo nagrade in posledično za ponovno izbiranje akcij, je pomembno, da skušamo simulacijsko strategijo čim bolj optimizirati za podano domeno. Pogosto vpeljemo t.i. hevristična pravila. To so pravila, ki so se v preteklosti eksperimentalno pokazala za najbolj uspešna. Iz naključnega izbiranja preidemo v t.i. psevdo naključno izbiranje akcij.

Simulacija se zaključi, ko zmanjka časa, (v kolikor imamo na razpolago omejeno količino časa), ali ko dosežemo končno stanje igre.

2.1.4.4 Vzratni prenos

Rezultat simulacije se aplicira na obiskana vozlišča. S tem zagotovimo, da je rezultat simulacije shranjen za nadaljnje iteracije. Lahko rečemo, da ima vzratni prenos vlogo spomina v algoritmu MCTS.

Zelo enostavna in učinkovita metoda je izračun povprečja nagrade glede na število obiskov vozlišča. V metodi nastopa:

- vrednost i -tega vozlišča drevesa V_i ,
- skupna nagrada vseh simulacij iz i -tega vozlišča R_i ,
- ter skupno število simulacij iz i -tega vozlišča N_i .

Vrednost vozlišča izračunamo:

$$V_i = \frac{R_i}{N_i}$$

Vrednost posameznega vozlišča je enaka razmerju med seštevkom nagrade in številom simulacij, ki so vključevale dotično vozlišče.

2.2 Texas Hold'em

Texas Hold'em spada v družino iger pokra. To je družina iger s kartami, ki temeljijo na večinoma enakih pravilih in enotnem principu vrednotenja kart [12]. Poznamo veliko število variacij pokra, vendar pa je v zadnjih letih različica Texas Hold'em postala najpopularnejša med njimi.

Zanimanje za igro se je povečalo okrog leta 2000, predvsem zaradi izpostavljenosti na televiziji, internetu in v popularni literaturi. K priljubljenosti igre veliko prispeva tudi enostavnost njenih pravil. Prav zaradi popularnosti in enostavnosti v zadnjih letih postaja predmet raziskav na področju strojnega učenja.

2.2.1 Opis igre

Za igro Texas Hold'em (v nadaljevanju Hold'em) sta potrebna vsaj dva igralca. Poleg samih kart (standardnih 52 kart) je potrebno imeti kot obvezni rekvizit še način za vrednotenje posameznih stav – običajno žetone, ki predstavljajo denar. Pri Hold'em igralec dobi 2 osebni karti in največ 5 skupnih kart, katere souporablja z vsemi ostalimi igralci. Iz največ 7 kart, ki jih ima na razpolago, mora sestaviti najboljšo možno kombinacijo.

Preden nadaljujemo z opisom igre, je potrebno definirati nekaj specifičnih pojmov [13]. V Tabeli 1 je predstavljeno osnovno izrazoslovje. Nekateri pojmi (Tabela 3) nimajo slovenskih izrazov, zato so navedeni v angleščini.

Tabela 1: Osnovni pojmi Texas Hold'em.

Pojem	Pojasnilo
Delilec	Delitelj kart. Gre za osebo, ki je v določeni igri zadolžena za delitev kart. Vloga delilca se po vsaki igri premakne za enega igralca naprej v smeri

	urinega kazalca (na levo stran trenutnega delilca).
»Pot«	Skupek vseh dosedanjih stav v določeni igri. Na koncu igre »pot« dobi zmagovalec, (oz. si ga razdelijo zmagovalci, v kolikor jih je več).
Velika slepa stava	Obvezna stava, ki jo mora vplačati igralec, ki se nahaja dve mesti levo od delilca. Njen namen je preprečiti, da bi igralci sedeli za mizo in igrali samo takrat, ko imajo dobre karte. Velika slepa stava se s časom veča in s tem dejansko prisili igralce v igro.
Majhna slepa stava	Polovica velike slepe stave. Ima isti namen kot velika stava. Vplača jo igralec levo od delilca.
Krog	Vsaka igra je sestavljena iz največ 4 krogov. Vsakič, ko igralci vidijo dodatno karto, se začne nov krog. V vsakem krogu (Tabela 3) se stave odvijajo neodvisno od ostalih krogov.
Roka	Najboljša kombinacija 5 kart, ki jo lahko posamezni igralec sestavi iz osebnih in skupnih kart.

V Tabeli 2 bom predstavil, katere možnosti ima igralec med igro. Vse možnosti so povezane s stavami. Ko je igralec na vrsti, lahko izvede eno od naštetih akcij. Akcije so v posameznih krogih neodvisne.

Tabela 2: Akcije pri Texas Hold'em.

Akcija	Pojasnilo
Izenačiti	Izenačitev stave. Igralec izenači trenutno najvišjo stavo v krogu. To je minimalni pogoj za napredovanje v naslednji krog.
Odstopiti	Odstop iz kroga. Igralec odstopi iz trenutnega kroga in iz trenutne igre. S tem izgubi ves denar, ki ga je do sedaj vložil v pot.
Klicati	Nadaljevanje brez stave. Poseben primer akcije »izenačiti«. Mogoče je samo, če v trenutnem krogu še ni nobene stave.
Staviti	Postavitev stave. Igralec postavi začetno stavo v trenutnem krogu.
Višati	Povečava stave. Igralec poviša stavo v trenutnem krogu.

Igra Hold'em ima 4 specifična stanja oziroma kroge, ki jih ločimo glede na stanje kart v igri (Tabela 3).

Tabela 3: Krogi pri Texas Hold'em.

Krog	Pojasnilo
»Pre-flop«	Prvi krog v vsaki igri. Igralec dobi svoje osebne karte.
»Flop«	Igralci dobijo 3 skupne karte.
»Turn«	Igralci dobijo 4. skupno karto.
»River«	Igralci dobijo 5. skupno karto.

Pogoj za prehod igre v naslednji krog je, da vsaj dva igralca izpolnujeta pogoje za prehod med krogoma. Za prehod med krogoma mora posamezni igralec izenačiti (klicati ali izenačiti) predhodno najvišjo stavo. V kolikor igralec nima dovolj denarja, se lahko odloči za »all in«.

To pomeni, da stavi ves svoj preostali denar in s tem lahko nadaljuje igro do konca. V kolikor zmaga, mu pripada sorazmerni delež »pota« (preostalo pripada drugouvrščenemu).

Igra se zaključi po končanem 4. krogu (»river«), oziroma ko niso več izpolnjeni pogoji za prehod v naslednji krog igre. V kolikor v igri ostane samo en igralec, je ta seveda zmagovalec. V nasprotnem primeru sledi vrednotenje kart igralcev, ki so ostali v igri. Igralec z najvišjo vrednostjo roke je zmagovalec.

2.2.1.1 Potek igre

Eden izmed igralcev za mizo je določen kot začetni delilec. Vrstni red igralcev določa njihov položaj za mizo. Igralca levo od delilca (v smeri urinega kazalca) v »pot« postavita veliko in majhno slepo stavo.

1. Začne se »pre-flop«. Razdelita se po dve osebni karti. Prvi na vrsti je igralec levo od igralca, ki mora postaviti veliko slepo stavo (3. igralec levo od delilca). Vsak igralec lahko izbere eno od prej opisanih akcij (Tabela 2). V kolikor so pogoji za prehod v naslednji krog izpolnjeni, se igra nadaljuje.
2. Začne se »flop«. Delilec razdeli tri skupne karte. Prvi na vrsti je igralec levo od delilca. Vsak igralec, ki je prišel v drugi krog, lahko izbere eno od prej opisanih akcij (Tabela 2). V kolikor so pogoji za prehod v naslednji krog izpolnjeni, se igra nadaljuje.
3. Začne se »turn«. Delilec razdeli četrto skupno karto. Prvi na vrsti je igralec levo od delilca. Vsak igralec, ki je prišel v tretji krog, lahko izbere eno od prej opisanih akcij (Tabela 2). V kolikor so pogoji za prehod v naslednji krog izpolnjeni, se igra nadaljuje.
4. Začne se »river«. Delilec razdeli peto skupno karto. Prvi na vrsti je igralec levo od delilca. Vsak igralec, ki je prišel v četrti krog, lahko izbere eno od prej opisanih akcij (Tabela 2). Igra se zaključi tako, da se ovrednoti karte igralcev, ki izpolnjujejo pogoje za prehod v naslednji krog, (čeprav naslednjega kroga ni). Določi se zmagovalec (ali v primeru izenačenja več zmagovalcev).

Obstajajo še različni robni pogoji, ki spremenijo potek igre, vendar za razumevanje osnovnega toka igre niso ključnega pomena. V kolikor bralec želi podrobnejše informacije [12], lahko poseže po eni izmed številnih knjig, ki so bile v zadnjem času izdane na temo pokra Hold'em.

2.2.2 Vrednotenje rok

Roka je sestavljena iz 5 kart. Izmed 7 kart, ki so na razpolago igralcu (2 osebni in 5 skupnih), se 2 odstranita tako, da igralec ostane z najboljšo možno roko. Imamo 10 različno močnih rok. Razvrstitev rok od najmočnejše (1.) do najšibkejše (10.) se nahaja v Tabeli 4 [4].

Tabela 4: Vrednotenje rok pri Hold'em.

Pozicija	Ime roke	Primer roke	Opis roke
1.	Kraljeva lestvica	T♠ J♠ Q♠ K♠ A♠	Vse karte po vrsti od 10 do asa v isti figuri.
2.	Barvna lestvica	2♠ 3♠ 4♠ 5♠ 6♠	Vse karte po vrsti v isti figuri.
3.	Poker	A♠ A♣ A♥ A♦ 5♠	Štiri karte enake vrednosti iz različnih figur.
4.	Polna hiša	A♠ A♣ A♥ K♠ K♣	En par in tri karte enake vrednosti v katerikoli figuri.
5.	Barva	A♠ 4♠ 7♠ 9♠ K♠	Vse karte določene figure.
6.	Lestvica	2♠ 3♣ 4♣ 5♥ 6♦	Karte po vrsti v različnih figurah.
7.	Tris	A♠ A♣ A♥ K♠ Q♣	Tri karte enake vrednosti iz različnih figur.
8.	Dva para	A♠ A♣ K♠ K♣ 3♥	Dva para kart, ne glede na figuro.
9.	En par	A♠ A♣ K♠ Q♣ 3♥	Dve karti enake vrednosti iz različnih figur.
10.	Najvišja karta	A♠ 3♣ 7♣ 9♥ Q♦	Nobena od zgoraj naštetih kombinacij. Šteje le, kdo ima v roki karto višje vrednosti.

Moč posameznih rok je pogojena z verjetnostjo njihovega pojava.

2.2.3 Stave

V vsaki igri so obvezne stave, t.i. majhna in velika slepa stava, ki ju ustrezna igralca položita v pot pred začetkom prvega kroga. Glede na omejitve pri vseh nadaljnjih stavah ločimo tri vrste Hold'em:

- »Limit Hold'em«: Zgodovinsko najbolj razširjena oblika v kazinojih in igrah v živo. Stava in višanje sta v prvih dveh krogih (»pre-flop« in »flop«) v višini velike slepe stave. V zadnjih dveh krogih (»turn« in »river«) pa sta v višini dvakratnika velike slepe stave.
- »Pot-limit Hold'em«: Največje možno višanje je enako trenutni vrednosti pota (vključujoč znesek, ki je potreben za klicanje).
- »No-limit Hold'em«: Stava in višanje sta v vseh krogih omejena samo navzdol. Minimalna stava ali višanje sta enaka maksimalni predhodni stavi ali višanju. Primer: V kolikor predhodni igralec stavi 4€, je minimalno višanje 8€ (4€ so potrebni za izenačenje obstoječe stave in 4€ znaša minimalno dovoljeno višanje).

Od same vrste pokra Hold'em sta zelo odvisna strategija in način igranja. »Limit Hold'em« je najbolj enostavna oblika, ker je višina stave določena vnaprej. S tem igro močno poenostavimo, ker nam pri odločitvah v igri ni več potrebno upoštevati višine stav nasprotnikov. Tudi v primeru, da naš agent stavi, je višina stave že vnaprej določena.

Zaradi zgoraj napisanega, »Pot-limit« in predvsem »No-limit« vpeljeta dodaten nivo kompleksnosti pri odločanju. Igralci lahko na potek igre vplivajo ne samo s svojimi kartami, ampak tudi z višino svojih stav.

2.2.4 Odločanje

Med igro igralci sprejemajo odločitve samo delno na podlagi verjetnosti zmage. Večjo težo pri odločanju o naslednji akciji ima intuicija - človeški faktor. Računalniki imajo s tovrstnim intuicijskim odločanjem veliko težav, zato se računalniški agenti večinoma odločajo na podlagi verjetnosti posameznih dogodkov. Število možnih rok je ključnega pomena pri determinističnem odločanju o najbolj dobičkonosni akciji v dani situaciji.

Tabela 5: Število možnih rok nasprotnikov v stanju »pre-flop« v odvisnosti od števila igralcev.

Število soigralcev	Število možnih rok
1	1.225
2	690.900
3	238.360.500
4	56.372.258.250
5	$\approx 9,7073 \times 10^{12}$
6	$\approx 1,2620 \times 10^{15}$

Z večanjem števila igralcev število možnih rok eksponentno narašča [11]. V Tabeli 5 imamo kot primer število možnih kombinacij zasebnih kart nasprotnikov na začetku igre. Kot lahko vidimo, število možnih začetnih stanj zelo hitro narašča s številom igralcev. Posledično je problem determinističnega odločanja vedno bolj kompleksen. Za tovrstno odločanje bi bilo pri večjem številu igralcev potrebno veliko procesorske moči (za sprotno računanje verjetnosti), ali veliko pomnilnika (za shranjevanje vnaprej izračunane verjetnostne tabele).

Število možnih iger pri pokru Texas Hold'em v načinu »heads up« (2 igralca) lahko izračunamo na sledeči način:

- V igri je 52 različnih kart.
- Na razpolago imamo 1326 različnih kombinacij dveh zasebnih kart.
- Posledično ima nasprotnik lahko $50 * 49/2 = 1225$ različnih rok.
- Skupno število različnih možnih iger je torej $1326 * 1225 = 1624350$.

Na zmago pri pokru pa poleg same verjetnosti v posamezni situaciji vpliva tudi način igre posameznika. Na primer, igralec lahko soigralce s previsoko stavu (glede na svojo roko) prepriča v to, da odstopijo, kljub temu, da imajo večjo verjetnost za zmago.

2.3 MCTS za Texas Hold'em

V kolikor želimo izdelati agent, ki za odločanje uporablja drevesno preiskovanje Monte Carlo, je potrebno definirati osnovne korake algoritma za specifično domeno [3,5]. S tem vpeljemo domensko znanje in tako omogočimo bolj učinkovito delovanje drevesnega preiskovanja.

Vsako stanje drevesa predstavlja eno stanje v igri in zahteva izvedbo natančno ene akcije. V posameznem krogu igre imamo lahko več stanj. Tak primer vidimo v Tabeli 6.

Tabela 6: Prikaz večjega števila stanj na krog.

Stanje	Krog	Igralec	Akcija	Opis
1.	1.	Agent	Klic	Agent igra brez stave.
2.	1.	Nasprotnik	Stava	Nasprotnik stavi določen znesek.
3.	1.	Agent	Izenačitev	Agent izenači znesek.
4.	2.	Agent	Klic	Začne se nov krog; agent igra brez stave.

Z večanjem števila stanj drevesa se večja tudi procesorska zahtevnost preiskovanja drevesa. Iz tega vidika je najugodnejša oblika pokra »Limit Hold'em«. V tej obliki je največje možno število stanj na krog $2n - 1$, kjer n predstavlja število aktivnih igralcev. V ostalih dveh oblikah je zaradi poljubne višine stav možnih veliko več stanj za posamezen krog.

Agent mora v vsakem stanju izbrati eno izmed akcij iz omejenega nabora (Tabela 2). Da bi drevesno preiskovanje poenostavil in pohitril njegovo delovanje, sem akcije združil v skupine:

- pasivna (Izenačitev in Klic),
- aktivna (Stava in Višanje),
- odstop.

2.3.1 Sestop

Preiskovalno drevo je sestavljeno iz stanj igre. Ker v igri sodelujeta vsaj dva igralca, stanje igre ni odvisno samo od našega agenta, ampak tudi od akcij nasprotnikov. Sestop zato ločimo na dva dela. Izbiramo akcije za lastna stanja in predvidevamo akcije nasprotnika v njegovih stanjih [3,5].

2.3.1.1 Izbira lastne akcije

Lastno akcijo izberemo na podlagi povprečne nagrade ciljnega stanja. V kolikor trenutno vozlišče ni bilo dovoljkrat obiskano, (vrednosti povprečnih nagrad stanj niso dovolj zanesljive), izberemo akcijo naključno [3,5].

2.3.1.2 Modeliranje nasprotnika

Z modeliranjem nasprotnika skušamo čim bolj natančno predvideti nasprotnikovo akcijo v danem stanju. Nasprotnikovo obnašanje je v večji meri odvisno od njegovih zasebnih in skupnih kart, v manjši meri pa od njegove igralne strategije. Na podlagi zgodovine akcij v trenutni igri lahko poskušamo ugotoviti verjetnostno porazdelitev zasebnih kart nasprotnika. Na podlagi zgodovine akcij in zasebnih kart v predhodnih igrah pa lahko ugibamo o nasprotnikovi igralni strategiji [5,14].

Cilj modeliranja je izbira čim bolj realne akcije v stanju nasprotnika.

2.3.2 Širitev

V vsaki iteraciji drevo razširimo za eno lastno akcijo. Ker v drevesu hranimo tudi stanja nasprotnikov, moramo v vsaki iteraciji drevo razširiti tudi z njihovimi predvidenimi akcijami [3,5]. Drevo dejansko razširimo za toliko stanj, kolikor je še aktivnih igralcev v igri.

Izbira lastne akcije pri razširitvi drevesa je naključna. Izbira akcij za nasprotnike upošteva njihov model.

2.3.3 Simulacija

Za simulacijo uporabimo svoje zasebne karte, verjetnostno porazdelitev kart nasprotnika, (v najslabšem primeru je to enakomerna porazdelitev), in naključno izbrane skupne karte. Lastne in nasprotnikove akcije so lahko naključno izbrane (enakomerna porazdelitev), ali izbrane na podlagi verjetnostne porazdelitve zasebnih kart. V slednjem primeru posamezni skupini akcij (pasivna, aktivna, odstop) dodelimo različne uteži glede na skupino zasebnih kart.

V procesu simulacije preizkušamo največje možno število kombinacij akcij z namenom, da bi ugotovili, katera nam prinaša največji potencialni dobiček [3,5].

Izgubljeno natančnost simulacije zaradi naključnega izbiranja akcij kompenziramo z velikim številom simulacij.

2.3.4 Vzratni prenos

Nagrada je v našem primeru izkupiček igre (simulacije).

$$\text{Nagrada} = \text{Prejeti denar} - \text{Vloženi denar}$$

Definirana je kot razlika med prejetim in vloženim denarjem. V kolikor v igri ne zmagamo, je nagrada seveda negativna.

Nagrado vedno apliciramo samo na lastna stanja. Obiskanim vozliščem povečamo števec obiskčnosti za 1. Nagrado apliciramo na vsa vozlišča, ki so bila obiskana v določeni igri [3,5]. To storimo tako, da za vozlišče izračunamo novo povprečno nagrado glede na število obiskov.

2.4 Parametri simulacije

Na podlagi teorije o Texas Hold'em in algoritmu MCTS je potrebno opredeliti parametre simulacij, ki so ključnega pomena za rezultat simulacij. Skušal bom ugotoviti vpliv različnih parametrov in implementacij na metrike, ki jih bom določil v nadaljevanju.

Glavni parametri preiskovanja MCTS, ki imajo ključni vpliv na izid simulacije, so:

- število iteracij algoritma MCTS,
- število simulacij na iteracijo algoritma MCTS,
- strategija izbiranja vozlišč,
- model nasprotnika,
- ohranjanje drevesa, oz. njegov prenos med različnimi iteracijami.

V nadaljevanju poglavja bom na kratko opisal, zakaj sem določen parameter označil kot pomemben za končni izid simulacije.

2.4.1 Število iteracij algoritma MCTS

To je eden glavnih parametrov, ki ima vsekakor največji vpliv od vseh. Parameter nam pove, kolikokrat se bo MCTS algoritem izvedel pred vsako odločitvijo. Tu ni dvoma, da večje število iteracij pomeni boljše odločanje. S številom iteracij determiniramo tudi čas odločanja. Običajno smo časovno, oziroma s procesorsko močjo omejeni na določen interval. Agent, ki se vedno optimalno odloči, vendar pa za vsako potezo porabi en dan, ni preveč uporaben v realnem življenju.

2.4.2 Število simulacij na iteracijo algoritma MCTS

Je tesno povezano s parametrom števila iteracij algoritma MCTS. Pove nam, koliko simulacij izvedemo za vsako MCTS iteracijo. Čeprav se zdi parameter enak številu iteracij algoritma

MCTS, je med njima majhna, a pomembna razlika. Razlika je toliko bolj opazna v okoljih z visoko varianco zunanjih parametrov. Nagrada, ki se aplicira na novo dodano vozlišče v vsaki iteraciji algoritma MCTS, je določena na podlagi rezultata simulacije. V kolikor simulacijo znotraj ene iteracije večkrat ponovimo, bo aplicirana nagrada bolj odražala dejansko vrednost poteze.

2.4.3 Strategija izbiranja vozlišč

Ta algoritem je ključnega pomena pri hitrosti konvergence k izbiri optimalne akcije. V kolikor bi imeli na razpolago neskončno število iteracij algoritma MCTS, bi bila lahko izbira akcije vedno naključna. Toda ker smo s številom simulacij omejeni, je ključnega pomena, da se osredotočimo na dele preiskovalnega drevesa, v katerih je predvidoma največja možnost najti optimalno akcijo.

2.4.4 Model nasprotnika

Dopolnjuje algoritem simulacije ter algoritem sestopa. Z njim simulacijo in sestop optimiziramo tako, da skušamo ugotoviti, kako se bo nasprotnik v določeni situaciji odzval, in s tem omejimo raziskovanje domene, ter povečamo možnost za izbiro optimalne akcije.

2.4.5 Ohranjanje drevesa med simulacijami

Da bi v predhodnih odločitvah pridobljeno znanje čim bolj izkoristili, imamo možnost to znanje prenesti na nadaljnje simulacije. Čeprav se to intuitivno vedno zdi optimalna rešitev, lahko pripelje tudi do nekaterih težav. Zaradi velike variance pri igri (naključne karte) se lahko poteza, ki je v prejšnji simulaciji veljala za zelo dobro, v trenutni simulaciji izkaže za zelo slabo.

3 Orodja

Na spletu lahko najdemo veliko izbiro ogrodij za razvoj agentov za igranje pokra. Večina je odprtokodnih. Zaradi poznavanja jezika sem se odločil za eno najbolj znanih ogrodij, napisanih v Javi – OpenTestBed. Uporablja ga veliko število raziskovalcev, zato na spletu najdemo veliko število nasprotnikov, s katerimi lahko primerjamo svoj agent. OpenTestBed je odprtokodna implementacija komercialnega produkta »Poker Academy«.

3.1 Poker Academy

Vodilno vlogo na področju umetne inteligence aplicirane na poker ima Raziskovalna skupina za poker (ang. Poker Research Group) iz Univerze v Alberti. Raziskovanje so začeli v sredini devetdesetih let. Njihov najbolj znan produkt je program »Poker Academy PRO« (Slika 5).



Slika 5: Program Proker Academy Pro.

Uporabnik lahko razvije svoj agent v Javi in s pomočjo ogrodja »Meerkat API« igra proti znanim agentom in agentom drugih uporabnikov.

Okolje omogoča učenje in spremljanje uspešnosti preko enostavnega in dodelanega grafičnega vmesnika. Prav tako je omogočen vpogled v celotno zgodovino odigranih rok, kar pripomore k enostavnosti razhroščevanja agentov.

Produkt »Poker Academy PRO« iz neznanih razlogov od konca leta 2011 ni več na voljo. Večina njegovih uporabnikov se je zaradi enostavnosti prehoda (isti programski vmesnik) preusmerila na OpenTestBed.

3.2 Meerkat API

»Meerkat API« je enostaven programski vmesnik, preko katerega se lahko različni agenti priključijo v skupno igro pokra [9]. Vsebuje vmesnike in razrede za vse osnovne elemente igre.

Za predstavo stanja igre strežnik uporablja osnovne razrede (Tabela 7), s katerimi opiše osnovne elemente igre in agenta.

Tabela 7: Osnovni razredi pri programskem vmesniku »Meerkat API«.

Razred	Opis razreda
Card	Razred predstavlja karto. Glavne lastnosti razreda sta Suit (barva) in Rank (vrednost karte).
Action	Razred opisuje akcijo, ki jo agent izvede med igro. Na voljo so vse akcije, ki jih Hold'em predvideva (Tabela 2). Pri akcijah, kjer je potrebno staviti (npr. višanje), se definira znesek v obliki spremenljivke tipa double.
GameInfo	Za dostop do trenutnega stanja igre agent uporablja razred GameInfo, ki mu ga strežnik poda ob začetku vsake igre. Preko razreda agent dostopa do stanja igre (pot, znesek denarja na razpolago, velikost velike slepe stave, krog itd.) in do akcij, ki jih ima trenutno na razpolago (višanje, odstop itd.).
Preferences	Ob vključitvi agenta v igro mu strežnik lahko poda nastavitve, ki jih določi razvijalec sam. Nastavitve so poljubne. Agent jih lahko uporabi za svoje interno delovanje. Za strežnik niso pomembne.

Strežnik komunicira z agenti preko vmesnika, na katerem so definirani vsi potrebni dogodki za spremljanje dogajanja za mizo. Strežnik izvršuje klice in nadzoruje izvajanje agenta. Agent mora le implementirati želeni odziv na podano situacijo.

Osnovni vmesniki, ki jih mora implementirati vsak agent, ki želi sodelovati v igri, so del razreda **com.biotoools.meerkat.Player** (Tabela 8).

Tabela 8: Osnovni dogodki pri agentu »Meerkat API«.

Metoda	Opis metode
void gameStartEvent (GameInfo gInfo)	Ob začetku nove igre strežnik agentu poda objekt GameInfo, s katerim lahko agent dostopa do stanja trenutne igre (stave, možne akcije, velikost pota itd.)
void holeCards (Card c1, Card c2, int seat)	Metoda se pokliče po delitvi zasebnih kart. Istočasno strežnik dodeli agentu tudi mesto za mizo.

Action getAction()	Strežnik pokliče akcijo na agentu, ko od njega zahteva odločitev o potezi. Zgodi se samo, ko je agent na vrsti.
---------------------------	---

Kljub temu, da so obvezne metode samo tiste, ki so opisane v Tabeli 8, mora vsak agent, ki želi imeti bolj celovit pogled na dogajanje v igri, implementirati še ostale metode v vmesniku (Tabela 9).

Tabela 9: Dodatni dogodki za agent »Meerkat API«.

Metoda	Opis metode
void init (Preferences playerPrefs)	Ob inicializaciji agenta mu strežnik poda parametre delovanja v obliki objekta Preferences.
void stageEvent (int stage)	Strežnik pokliče proceduro na agentu ob začetku vsakega kroga (0-3).
void showdownEvent (int seat, Card c1, Card c2)	Metoda se izvede ob dogodku, ko agenti predstavijo svoji zasebni karti. Seat predstavlja pozicijo agenta, Card c1 in Card c2 pa njegovi zasebni karti.
void dealHoleCardsEvent ()	Strežnik sporoči agentu, da je vsem igralcem razdelil zasebne karte.
void actionEvent (int pos, Action act)	Akcija se je zgodila za mizo. Dogodek se sproži, ko eden od agentov odgovori na proceduro getAction() .
void gameStateChanged ()	Strežnik sporoča agentu, da se je GameInfo objekt spremenil, in s tem agentu omogoča dostop do novega stanja objekta. To se zgodi po vsaki uspešno izvedeni akciji s strani igralcev.
void gameOverEvent ()	Dogodek zaznamuje konec igre.
void winEvent (int pos, double amount, String handName)	Strežnik vsem igralcem sporoči, kdo je zmagovalec v trenutni igri. Pos predstavlja pozicijo igralca za mizo, amount znesek, za katerega je zmagal in handName roko, s katero je zmagal.

3.2.1 Primer igre

Na Sliki 6 je podan primer igre agenta na strežniku, ki podpira »Meerkat API«.

```
/* Strežnik pokliče agent in mu poda konfiguracijski objekt */
Init(Preferences preferences);

/* Strežnik agentu sporoči, da se je pričela nova igra */
gameStartEvent(GameInfo gameInfo);

/* Strežnik agentu dodeli zasebne karte in mesto za mizo */
holeCards(Card card, Card card2, int i);

/* Strežnik agentu sporoči, da je razdelil zasebne karte */
dealHoleCardsEvent();

/* Strežnik agentu sporoči postavitev velike in male slepe stave */
actionEvent (int i, Action action);
actionEvent (int i, Action action);
```

```

/* Strežnik agentu sporoči prehod v novi krog igre */
stageEvent(int i);

...Se ponavlja...

/* Strežnik pokliče agent in od njega zahteva akcijo */
Action getAction ();

/* Strežnik agent obvesti o akciji nasprotnika */
actionEvent (int i, Action action);
...

/* Strežnik agentu sporoči konec igre in pokaže karte nasprotnikov */
gameOverEvent();
showdownEvent(int i, Card card, Card card2);

/* Strežnik agentu sporoči zmagovalca igre */
winEvent(int i, double v, String s);

```

Slika 6: Primer igre agenta, ki podpira programski vmesnik »Meerkat API«.

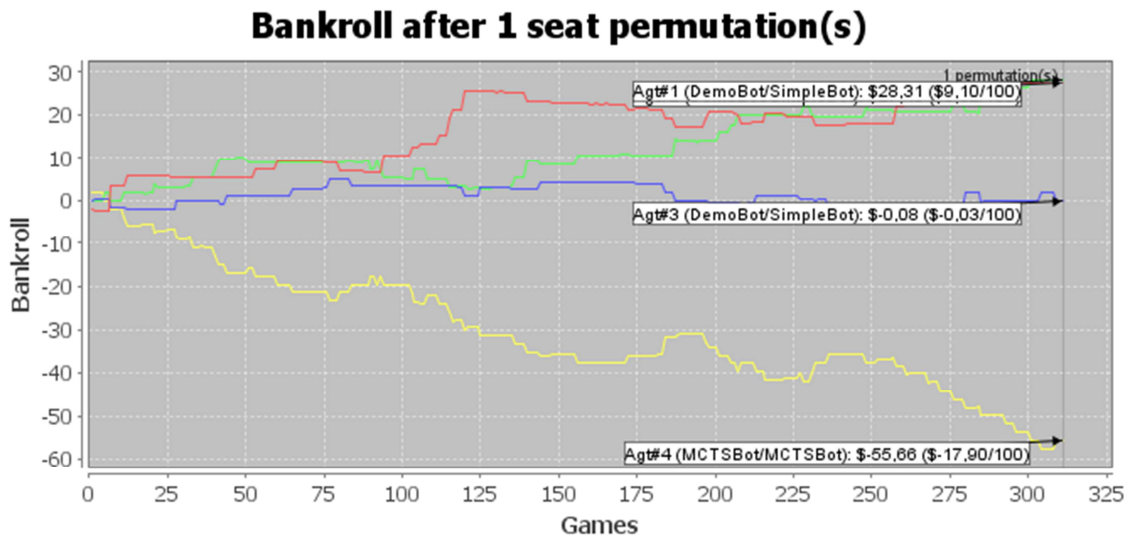
3.3 OpenTestBed

OpenTestBed je testno okolje za agente napisane s programskim vmesnikom »Meerkat API«. Predstavlja polno implementacijo vmesnika, zato omogoča igranje tudi najbolj naprednim agentom. Program je odprtokoden in prosto dostopen na spletu. Poleg implementacije samega programskega vmesnika omogoča še številne funkcionalnosti, ki nam olajšajo analizo in grafični prikaz rezultatov [9].

Ključne funkcionalnosti, ki jih potrebujemo za analizo rezultatov agenta, so:

- Simulacije poljubnega števila igralcev za »limit« in »no-limit« igre: Ugotovimo lahko, kako se naš agent obnaša v različnih igrah s spreminjajočim se številom nasprotnikov.
- Simulacije turnirjev: Ključnega pomena je obnašanje agenta v različnih delih turnirja.
- Uvoz predhodnih rezultatov agenta: Z uvozom že obstoječih podatkov o agentu učenja ni potrebno pričeti vedno od začetka.
- Simulacija več iger z enakimi kartami: Zmanjšuje varianco, oz. element sreče pri igri, in s tem omogoča bolj objektivno evaluacijo agenta.
- Izpis podatkov v tabelarični obliki po končani simulaciji: Podatke lahko uvozimo v program za njihovo obdelavo in analizo.
- Izris podatkov v grafični obliki v realnem času (med simulacijo): Uspešnost agenta lahko spremljamo že med samo simulacijo.

Grafični prikaz (Slika 7), ki ga bom v diplomski nalogi največ uporabljal, je graf finančnega stanja (ang. bankroll) agenta.



Slika 7: Grafični prikaz finančnega stanja agentov skozi igro.

Na X osi se nahaja podatek o zaporednih rokah, na Y osi pa podatek o trenutnem finančnem stanju posameznega agenta. Iz grafa lahko razberemo trend finančnega stanja našega agenta in s tem predvidimo njegovo donosnost na večje število iger. Varianca donosnosti je lahko velika, zato je za natančno določanje trenda potrebno veliko število odigranih rok. Pri mojih analizah bom uporabil minimalno 4000 rok. Iz izvedenih testov namreč izhaja, da se trend po 4000 rokah ne spreminja več bistveno.

3.4 Obstoječi agenti

Skupaj s testnim okoljem OpenTestBed dobimo tudi nekaj agentov [9], s katerimi učimo in primerjamo lastnega agenta. Agenti so med seboj zelo različni, in sicer od najbolj trivialnih (AlwaysCallBot) do najbolj naprednih (Chumpbot).

Za učenje in vrednotenje svojega agenta sem si izbral štiri nasprotnike:

1. AlwaysCallBot
2. FlockBot
3. SimpleBot
4. Chumpbot

Ker se agenti med seboj zelo razlikujejo, to močno vpliva na rezultate simulacije. Da bi si bralec lažje predstavljal igro in znal bolje interpretirati rezultate simulacij, bom za vsakega agenta, ki je nastopal kot nasprotnik, opisal osnovni način delovanja. Razvrščeni so od najbolj trivialnega do najbolj naprednega. V takem vrstnem redu sem tudi dejansko izvedel teste. Vsak izmed agentov prispeva višji nivo sofisticiranosti igre in zvišuje težavnost odločanja.

3.4.1 AlwaysCallBot

Kot že samo ime agenta namiguje, je njegova implementacija trivialna. Agent namreč pozna samo eno akcijo in to uporabi ne glede na okoliščine. Akcija je »kliči«, kar dejansko pomeni, da bo agent igral vsako roko, ne glede na karte in stave [9].

Uporaben je za osnovno vrednotenje lastnega agenta. Tukaj ne gre za dejansko igro proti nasprotniku, ampak zgolj za analizo pravilnosti delovanja pri trivialnih odločitvah v igri. Preizkusiti želim, ali se moj agent pravilno obnaša pri enostavnih odločitvah.

3.4.2 FlockBot

Agent igra prve 3 kroge na enak način kot AlwaysCallBot. V 4. krogu (ang. river) se agent odloča na podlagi statične matematične formule:

- ovrednoti vrednost svoje roke na podlagi kart in števila igralcev,
- vsakemu igralcu določi moč roke samo na podlagi števila igralcev,
- v kolikor je moč njegove roke višja od moči roke nasprotnikov:
 - o izenači, v primeru da je nekdo v zadnjem krogu že stavil,
 - o v nasprotnem primeru viša; znesek višanja je enak polovici trenutnega pota,
- če moč njegove roke ni višja od moči roke nasprotnikov, odstopi.

Za vrednotenje svoje roke uporablja knjižnico HandEvaluator (prisotna v »Meerkat API«), ki na podlagi zasebnih kart, skupnih kart in števila igralcev vrne vrednost med 0 in 1. Ta vrednost predstavlja moč roke. Za vrednotenje roke nasprotnikov uporablja enostavno formulo $x = 1 - 1/n$, kjer je x moč roke in n skupno število aktivnih igralcev v roki [10].

Zame je njegovo obnašanje pomembno, ker z njim lahko testiram obnašanje lastnega agenta pri enostavnih stavah nasprotnika.

3.4.3 SimpleBot

Tudi ta agent ima samo dva načina delovanja, in sicer na podlagi vnaprej določene logike v 1. krogu, ter na podlagi verjetnosti in stav v ostalih krogih [9]. Vendar pa je njegova implementacija v primerjavi z agentom FlockBot bolj napredna.

Logika delovanja v 1. krogu:

- Igraj, v kolikor sta zasebni karti par.
- Igraj, v kolikor sta obe zasebni karti višji ali enaki 10.
- Igraj, v kolikor sta zasebni karti iste barve in se njuna vrednost razlikuje za 1 (ang. suited connectors).
- Igraj, v kolikor sta zasebni karti iste barve in je ena izmed njih as.
- Igraj v 5% primerov, ko zgoraj naštetih pogoji niso izpolnjeni.

Logika delovanja v ostalih krogih:

- Izračunaj, kolikšen je delež vložka v primerjavi s potom.
- Izračunaj moč svoje trenutne roke.
- Izračunaj približek potenciala roke.
- Na podlagi izračunanega potenciala roke:
 - o višaj, oz. stavi (največ možno),
 - o v kolikor ni še nihče stavil, stavi (proporcionalno glede na moč roke),
 - o izračunaj, katera akcija (med »kliči«, »višaj« in »odstopi«), je najbolj ugodna glede na izračunan potencial roke.

Kot je razvidno iz opisa, gre za prvega pravega nasprotnika mojemu agentu. Čeprav je enostaven, se SimpleBot dobro obnese proti večini nasprotnikov, saj je njegovo delovanje v celoti osnovano na verjetnosti. Z njim testiramo uspešnost lastnega agenta pri konstantni strategiji nasprotnika.

3.4.4 ChumpBot

Ima dva osnovna načina delovanja, tako kot SimpleBot. Od njega se razlikuje v tem, da je prvi izmed agentov, ki svojo strategijo igranja spreminja med igro. Strategija je definirana na podlagi dveh parametrov, in sicer poguma in drznosti. Obe vrednosti sta diskretni in lahko zavzameta eno izmed 9 vnaprej določenih vrednosti. Strategijo spremeni, oziroma naključno določi nove vrednosti spremenljivk v 10% primerov, ko roko izgubi. Parametra omogočata agentu skupaj 81 različnih strategij igranja [10].

Logika delovanja v 1. krogu:

- Igraj, v kolikor sta zasebni karti višji od 10.
- Igraj, v kolikor se zasebni karti razlikujeta za manj kot 3 (večja možnost za lestvico).
- Igraj, v kolikor zasebni karti nista različne barve (večja možnost za barvo).
- V kolikor noben od zgoraj naštetih pogojev ni izpolnjen, igray samo, če ni še nihče od nasprotnikov stavil (akcija izenači).

Logika delovanja v ostalih krogih:

- Na podlagi vrednosti pota, količine lastnega denarja in višine slepe stave določi spremembo poguma in drznosti.
- Na podlagi poguma, drznosti in kroga določi trenutno moč.
- Na podlagi poguma določi stopnjo blefiranja.
- Izračunaj vrednost svoje roke.
- V kolikor sta v trenutnem krogu potekli že več kot 2 agresivni akciji, igray pasivno.
- S pomočjo izračunane vrednosti lastne roke, stopnje blefiranja in trenutne moči se odloči za akcijo in višino morebitne stave.

ChumpBot spada v skupino agentov, ki svojo strategijo med igro naključno spreminjajo. S tem se stopnja zahtevnosti igre močno poveča. V kolikor želimo na veliko število iger premagati ChumpBota, se mora agent aktivno prilagajati njegovi strategiji. Z njim testiram prilagodljivost svojega agenta na različne strategije nasprotnika.

4 Izvedba

Prva faza razvoja vsakega kompleksnega sistema je posvečena načrtovanju sistema in postopkov testiranja. Iz izkušenj vemo, da tak pristop zmanjša tveganje za napake in posledično precej skrajša čas razvoja.

4.1 Ideja o izvedbi

Pred začetkom dela so bili vsi potrebni koraki razdeljeni na več vsebinskih sklopov. Sklopi so bili izvedeni zaporedno v naslednjem vrstnem redu:

1. Izdelava drevesnega preiskovanja Monte Carlo.
2. Izdelava agenta za igranje pokra Hold'em.
3. Izdelava načinov za zajemanje in prikazovanje podatkov.
4. Izdelava načrta izvajanja poskusov.
5. Priprava primerjalnih podatkov iz obstoječih agentov.

Razdelitev dela na samostojne sklope je omogočila ločeno testiranje in evaluacijo vsakega sklopa z namenom, da se izognem kompleksnim težavam, ki bi lahko zajemale več sklopov istočasno. Vsak sklop nadgrajuje predhodnega tako, da imamo po končanem zadnjem sklopu delujočo programsko rešitev in načrt izvajanja poskusov.

V nadaljevanju bom opisal osnovne ideje, ki sem jih uporabil pri implementaciji posameznega sklopa.

4.1.1 Izdelava drevesnega preiskovanja Monte Carlo

Izdelati je potrebno algoritem za drevesno preiskovanje, ki nam bo omogočal enostavno testiranje vpliva različnih parametrov in pristopov na rezultat simulacije. Zaradi obširnosti domene in visoke variance parametrov (naključne karte, nepredvidljivo obnašanje nasprotnika itd.), je potrebno algoritem čim bolj optimizirati, da bi tako skrajšali čas simulacij.

4.1.2 Izdelava agenta za poker Texas Hold'em

Agent za igranje Hold'em mora za svoje odločanje uporabljati drevesno preiskovanje Monte Carlo. Omogočati mora enostavno spreminjanje parametrov simulacije. Iz samega ogrodka

OpenTestBed, v katerem se bo agent izvajal, mora sprejemati stanje igre in akcije ostalih agentov. Beležiti mora zgodovino svojih odločitev za namene analize delovanja.

4.1.3 Izdelava načinov za zajemanje in prikazovanje podatkov

Agent mora za vsako akcijo v tekstovno datoteko zapisati vsaj prvo stopnjo listov drevesa, (to so listi, ki izhajajo neposredno iz korena drevesa), da lahko preverimo, kakšen je pričakovani dobiček za posamezno akcijo. Poleg tega mora v tekstovno datoteko zapisati še svoji zasebni karti, skupne karte, trenutno stavo in izbrano akcijo. Na tak način lahko preverjamo pravilnost odločanja v enostavnih primerih (npr. v primeru zelo močne ali zelo šibke roke).

Ogrodje OpenTestBed že omogoča spremljanje finančnega stanja posameznega agenta, zato funkcionalnosti ne bo potrebno implementirati.

4.1.4 Izdelava načrta izvajanja poskusov

Primerjave bodo potekale na vzorcu 4000 iger, in sicer z vnaprej generiranimi in permutiranimi kartami. To pomeni, da so karte za igro vnaprej znane, in da oba igralca dobita vse možne kombinacije rok. Posledično imata enake možnosti za zmago.

Kot osnovno metriko uspešnosti bom vzel končni dobiček, in sicer z igro proti agentu AlwaysCallBot. Naiven agent nam ponuja možnost vrednotenja lastne implementacije brez variance, ki jo povzroča agent z naprednejšim modelom odločanja.

Osnovna simulacija, ki jo bomo skušali izboljšati s spreminjanjem pomembnih parametrov, je definirana v Tabeli 10.

Tabela 10: Parametri osnovne simulacije.

Parameter simulacije	Vrednost
Število iteracij algoritma MCTS	10000
Število simulacij na iteracijo algoritma MCTS	1
Strategija izbiranja vozlišč	Naključna izbira akcije
Model nasprotnika	Naključna izbira
Ohranjanje drevesa	Drevo se v vsaki simulaciji gradi od začetka

Med simulacijami bom naenkrat variiral samo en parameter z namenom določiti njegov vpliv na rezultat simulacije. Parametre bom vedno variiral nad osnovnimi simulacijami. Na koncu bom izvedel še simulacije z najboljšo kombinacijo vseh parametrov.

Izjema je parameter »Število iteracij algoritma MCTS«. Z večanjem števila iteracij se uspešnost agenta izboljšuje (ne glede na ostale parametre). Ker ima parameter velik vpliv na čas izvajanja simulacije, ga bom omejil na konfiguracije opisane v Tabeli 11.

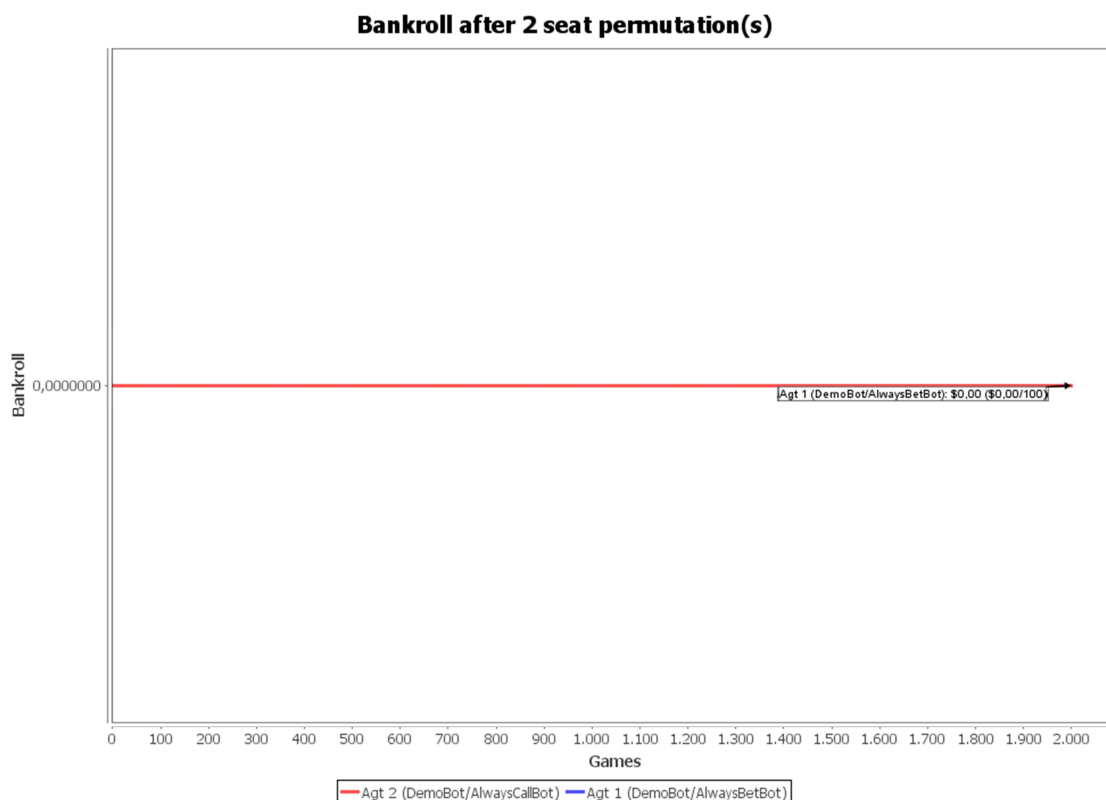
Tabela 11: Konfiguracije števila iteracij algoritma MCTS.

Ime konfiguracije	Število iteracij algoritma MCTS
Nizko število iteracij	5000
Srednje število iteracij	10000
Visoko število iteracij	15000

4.1.5 Priprava primerjalnih podatkov

Zaradi načina izvajanja poskusov (vnaprej generirane in permutirane karte) je izkupiček determinističnega agenta, ki igra sam proti sebi, vedno 0 (Slika 8). Ravna črta na grafu zagotavlja, da bodo vse simulacije neodvisne od samih kart.

Za metriko končne uspešnosti bom uporabil najbolj učinkovit, pretežno deterministični agent SimpleBot. Čeprav je agent ChumpBot naprednejši, sem se odločil za agent SimpleBot, ker ta nima različnih načinov delovanja (ne blefira). Blefiranje v primeru nasprotnika, ki igra vedno enako, pripelje do manj optimalne igre in posledično do nižje uspešnosti.



Slika 8: Prikaz dobička determinističnega agenta, ki igra sam s seboj.

4.2 Rezultati izvedbe

Po implementaciji v predhodnem poglavju opisanih idej smo prišli do ogrodja za algoritem MCTS in vrsto prikazovalnikov, ki nam omogočajo spremljanje in razhroščevanje delovanja algoritma. V sklopu izdelave agenta za igranje pokra Hold'em je bilo potrebno razviti tudi simulator igre poker Hold'em.

Na podlagi implementacije orodij in načrta za izvajanje testov je bil sestavljen končni nabor parametrov in poskusov testiranja.

4.2.1 Ogrodje MCTS

Drevesno preiskovanje Monte Carlo je izdelano v obliki modularnega ogrodja, napisanega v Javi. Glavne elemente ogrodja lahko zlahka zamenjamo brez spreminjanja obstoječe kode. To nam omogoča fleksibilnost pri iskanju najboljše rešitve za dano domeno.

Ogrodje razvijalcu omogoča abstrakcijo in izmenjevanje naslednjih delov algoritma:

- list drevesa: definiramo lahko svoj objekt, ki hrani različne podatke,
- akcije v domeni: definiramo domensko specifične akcije,
- prikazovalnik: podatke v drevesu lahko prikazujemo na različne načine,
- algoritem za izbiro akcije,
- algoritem za razširitev drevesa,
- algoritem za simulacijo igre,
- algoritem za aplikacijo nagrade,
- model nasprotnika.

Drevesno preiskovanje deluje po algoritmu, opisanem v poglavju 2.1.4. Performančno je ogrodje za izbrano domeno (Hold'em poker) ustrezno, saj omogoča približno 5000 simulacij na sekundo.

Skupaj z ogrodjem je bil izdelan tudi nabor osnovnih algoritmov, potrebnih za delovanje drevesnega preiskovanja Monte Carlo:

- List drevesa:
 - o osnovni list drevesa, ki vsebuje vrednost in število obiskov.
- Izbira akcije:
 - o naključna izbira akcije,
 - o izbira akcije UCT.
- Razširitev drevesa:
 - o naključna razširitev drevesa za en list.
- Aplikacija nagrade:
 - o povprečna nagrada glede na število obiskov.
- Model nasprotnika:

- o igranje proti samemu sebi,
- o naključni model nasprotnika.

Za potrditev pravilnega delovanja ogrodja so bili izdelani tudi testni scenariji. Poleg tega pa je v ogrodje vključen tudi primer enostavne implementacije agenta in prikazovalnikov za igro križec krožec (ang. tic-tac-toe).

Ogrodje z izvorno kodo je javno dostopno na naslovu:

<https://bitbucket.org/Andrejj/mcts-framework/>

4.2.2 Poker agent za »Meerkat API«

Z uporabo ogrodja za MCTS in programskega vmesnika »Meerkat API« sem izdelal agent za igranje Hold'em. Agent lahko vključimo v kateri koli strežnik, ki je kompatibilen z »Meerkat API«. Agent vodi tako zgodovino lastnega pota, kot tudi zgodovino kart in akcij.

Za lažje izvajanje testov ima agent konfiguracijsko datoteko, v kateri lahko specificiramo vse parametre agenta in simulacije brez spreminjanja izvorne kode.

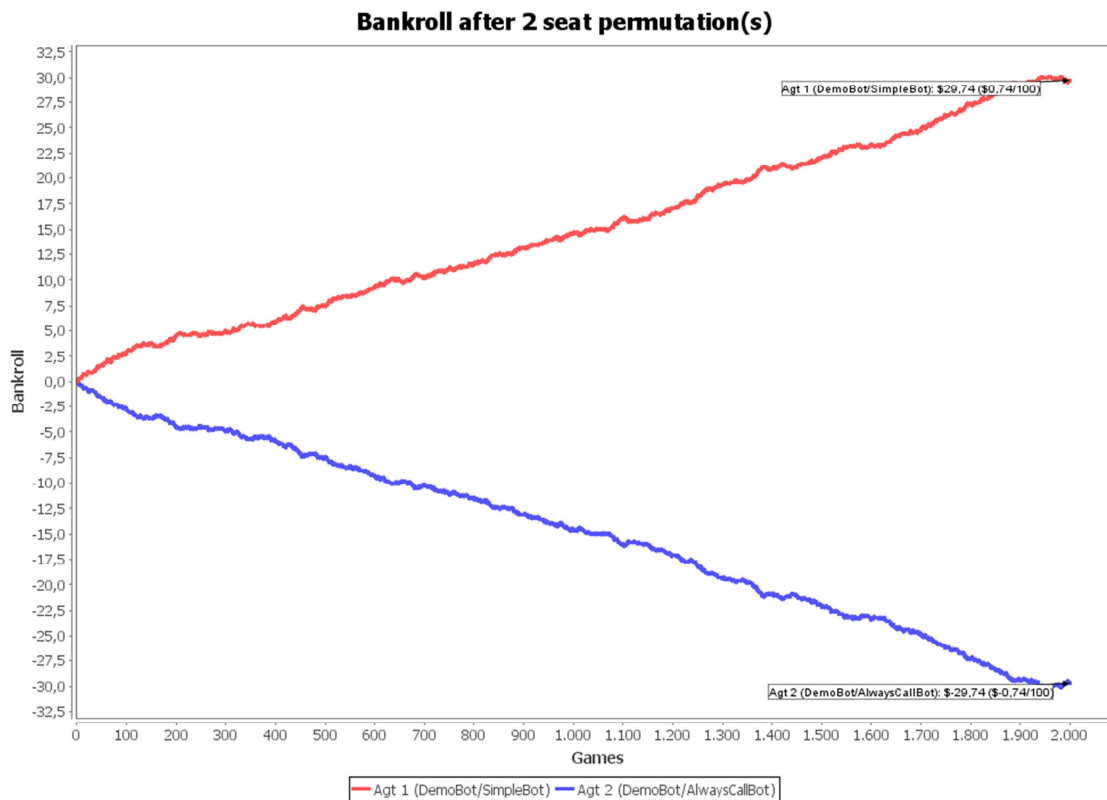
Agent je bil posredovan ekipi, ki vzdržuje OpenTestBed z namenom, da ga vključijo v eno od naslednjih verzij orodja.

4.2.3 Pridobljeni podatki

Osnovni rezultat, s katerim sem primerjal svoj agent, je igra SimpleBot proti agentu AlwaysCall. V Tabeli 12 vidimo končne rezultate igre. Na Sliki 9 je grafično pokazan potek dobička agentov skozi igre.

Tabela 12: Prikaz učinkovitosti agenta SimpleBot proti agentu AlwaysCallBot.

Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/ veliko slepo stavo
27,94	0,0069	0,34



Slika 9: Prikaz dobička igre agenta SimpleBot proti agentu AlwaysCallBot.

Skupni dobiček agenta SimpleBot po 4000 igrh je bil 27.94, kar znaša 0.007 dobička na igro. Ker je višina dobička odvisna izključno od višine velike slepe stave, je smiselno navajati vrednost koeficienta povprečnega dobička na igro v primerjavi z veliko slepo stavo. Ta znaša za naš primer 0.34, kar pomeni, da v vsaki igri agenta SimpleBot povprečno pridobimo 34% velike slepe stave. Iz grafa je tudi razvidno, da dobiček večinoma konstantno raste ne glede na karte, s katerimi agenti igrajo.

Da bi lahko proglasil svoj agent za uspešen, mora vsaj izenačiti koeficient povprečnega dobička na igro agenta SimpleBot. V nasprotnem primeru je nadaljnje raziskovanje nesmiselno, saj deterministični agent očitno opravi nalogo bolj optimalno.

4.2.4 Težave

Med samo izvedbo sem se srečal s številnimi težavami, povezanimi z orodjem OpenTestBed. Orodje je prirejeno za testiranje NoLimit Holdem igre z več igralci, zato kar nekaj specifik pokra Limit Hold'em ne deluje pravilno. Od leta 2011 orodje nima popravkov, zato je posledično prisotnih kar nekaj (tudi kritičnih) napak. Napake sem moral med samo izvedbo sproti popravljati, saj so onemogočale nadaljnje testiranje.

Nekaj najbolj kritičnih napak, ki sem jih odpravil:

- V določenih primerih, ko sta igralca na koncu igre izenačena, se zgodi deljenje z nič in posledično se simulacija ustavi.
- V »flop«, »turn« in »river« fazi je v »heads up« načinu igre na vrsti napačni igralec, in posledično se simulacija MCTS (ang. playout) odloča na podlagi napačnih podatkov.
- V posameznih primerih, ko je določen igralec v stanju »all in«, se napačno dodeli znesek zmage.

Zgoraj našteje napake so splošno znane, rešitve ali vsaj ideje za rešitev zanje pa je mogoče najti na spletu [9].

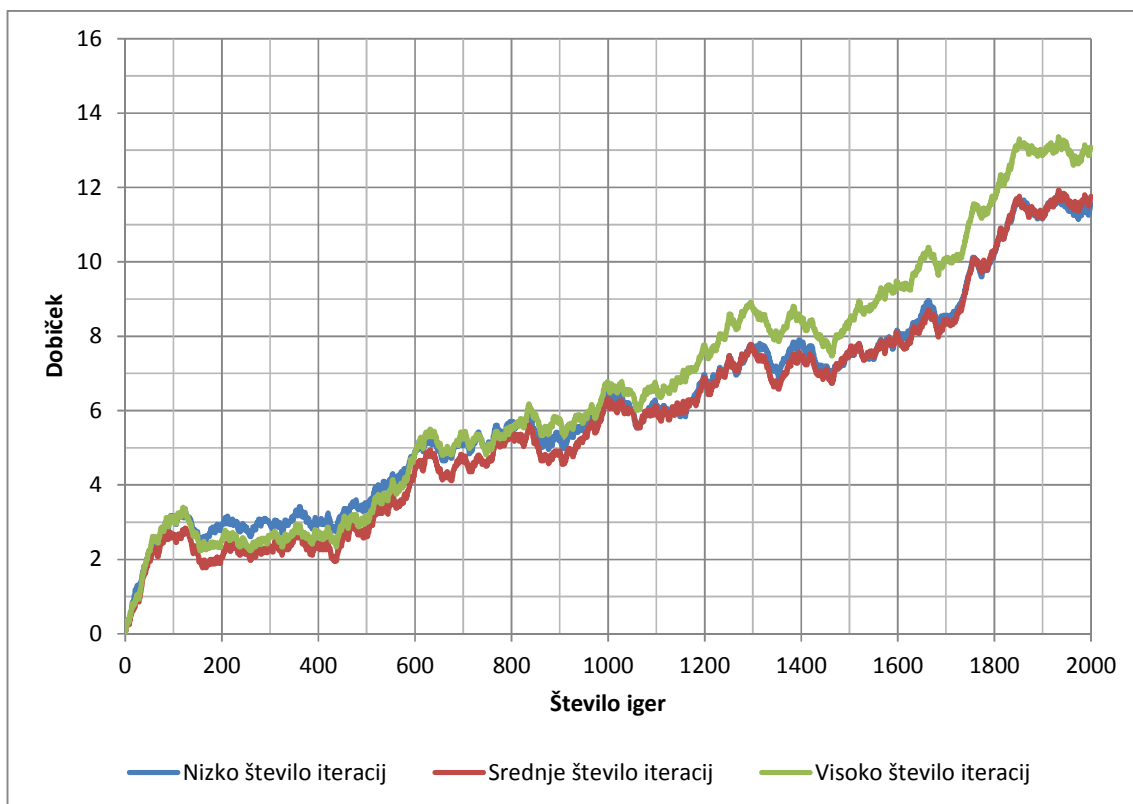
5 Rezultati

Osnovni rezultat, ki služi kot osnova za optimizacijo, je igra agenta proti agentu AlwaysCallBot. Sledijo podatki osnovnih simulacij izvedenih na podlagi parametrov, določenih v poglavju 4.1.4.

Tabela 13: Prikaz učinkovitosti osnovnega agenta MCTS proti agentu AlwaysCallBot.

Število iteracij algoritma MCTS	Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/veliko slepo stavo	Razlika povprečnega dobička/igro/veliko slepo stavo v primerjavi z agentom SimpleBot
5000	11,54	0,0028	0,14	-0,20
10000	11,76	0,0029	0,15	-0,19
15000	13,08	0,0032	0,16	-0,18

V Tabeli 13 lahko vidimo spreminjanje rezultatov simulacije na podlagi števila iteracij algoritma MCTS. Že na prvi pogled je očitno, da kljub trikrat večjemu številu iteracij med posameznimi simulacijami ni velike razlike pri končnem dobičku. Dejstvo gre pripisati velikosti domene in relativno nizkemu številu simulacij v primerjavi z možnimi stanji igre. Zaradi časovnih omejitev števila iteracij ni smiselno dalje večati, ampak se je bolje osredotočiti na izboljšavo preiskovalnega drevesa.



Slika 10: Graf poteka dobička osnovnega agenta MCTS proti agentu AlwaysCallBot

Na Sliki 10 lahko vidimo potek dobička posamezne simulacije. Razberemo lahko, da je potek (vzponi in padci) dobička pri vseh treh simulacijah zelo podoben. Iz tega sklepamo, da se algoritem ne obnaša naključno – njegovo delovanje je pogojeno z nekim kriterijem optimalnosti, h kateremu stremi agent.

Razlika v končnem dobičku simulacije nastane zaradi različnih odločitev v posameznih igrah, kjer je izid bolj neodločen – igralca imata približno enakovredne karte. Z večjem številom simulacij ima agent na razpolago več informacij za odločitev o naslednji potezi, zato je njegova odločitev bolj optimalna.

5.1 Analiza vpliva parametrov algoritma MCTS

V nadaljevanju bom predstavil rezultate eksperimentov vpliva parametrov algoritma MCTS.

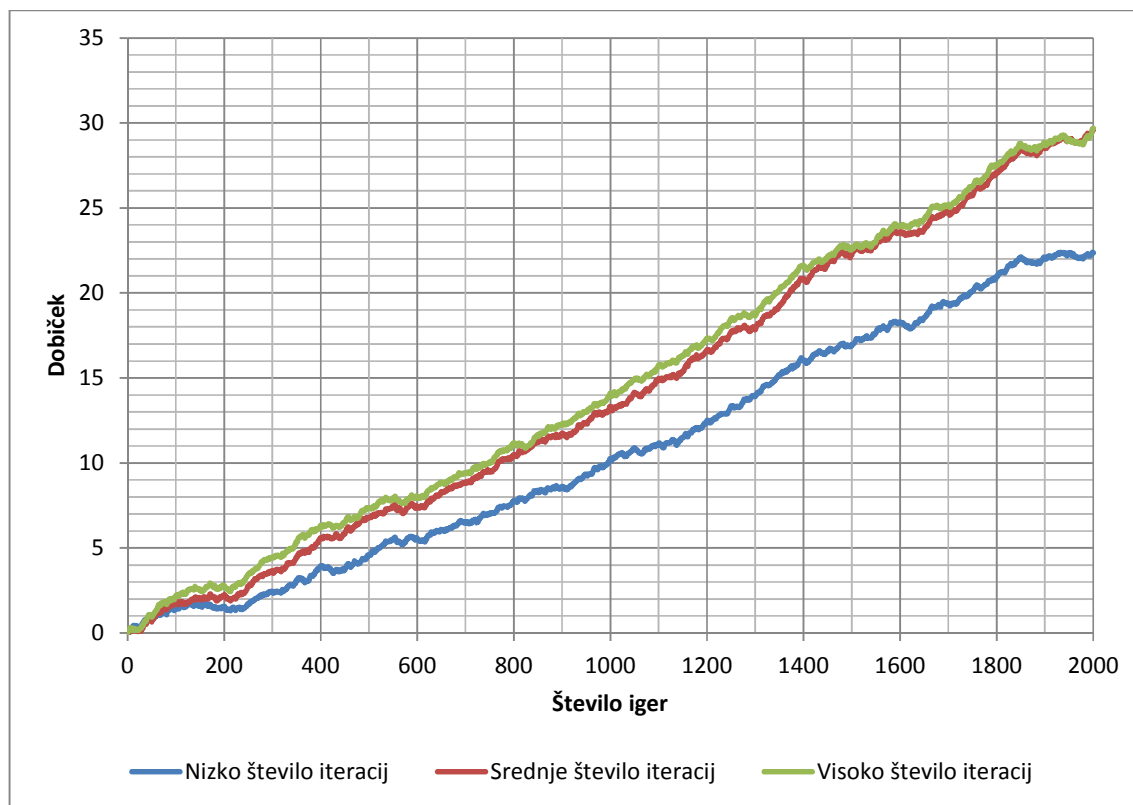
5.1.1 Strategija izbiranja vozlišč

Naključni algoritem za izbiro akcije sem zamenjal z algoritmom UCT. Rezultati zamenjave, kot tudi razlika glede na simulacije osnovnega agenta MCTS, so navedeni v Tabeli 14.

Tabela 14: Prikaz učinkovitosti agenta MCTS z izboljšanim algoritmom za izbiro akcije proti agentu AlwaysCallBot.

Število iteracij algoritma MCTS	Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/veliko slepo stavo	Razlika povprečnega dobička/igro/veliko slepo stavo v primerjavi z osnovnim agentom MCTS
5000	22,36	0,0056	0,28	+0,14
10000	29,58	0,0074	0,37	+0,22
15000	29,68	0,0074	0,37	+0,21

Bistveno izboljšanje rezultatov gre pripisati predvsem hitrejši konvergenci algoritma UCT k izbiri optimalne akcije v drevesu. Omejeno število simulacij, ki jih imamo na razpolago, porabimo na najbolj obetavnih vejah drevesa, in tako bolje raziščemo prostor, ki nam lahko prinese največ nagrade. Ker so simulacije bolj usmerjene, se večanje števila iteracij bolj pozna pri končnem rezultatu kot pri osnovnem agentu. Samo v simulaciji z nizkim številom iteracij (5000) agent nima boljšega rezultata kot deterministični agent SimpleBot.



Slika 11: Graf poteka dobička agenta MCTS z izboljšanim algoritmom za izbiro akcije proti agentu AlwaysCallBot.

Na Sliki 11 vidimo večjo učinkovitost agenta MCTS tudi z vidika konstantne rasti skupnega dobička. V primerjavi z osnovnim agentom MCTS je število padcev (in njihov razpon) veliko manjši – dobiček bolj konstantno raste. To kaže, da se agent bolj optimalno odloča tudi, ko ima slabše karte – igra pasivno, oziroma sploh ne igra.

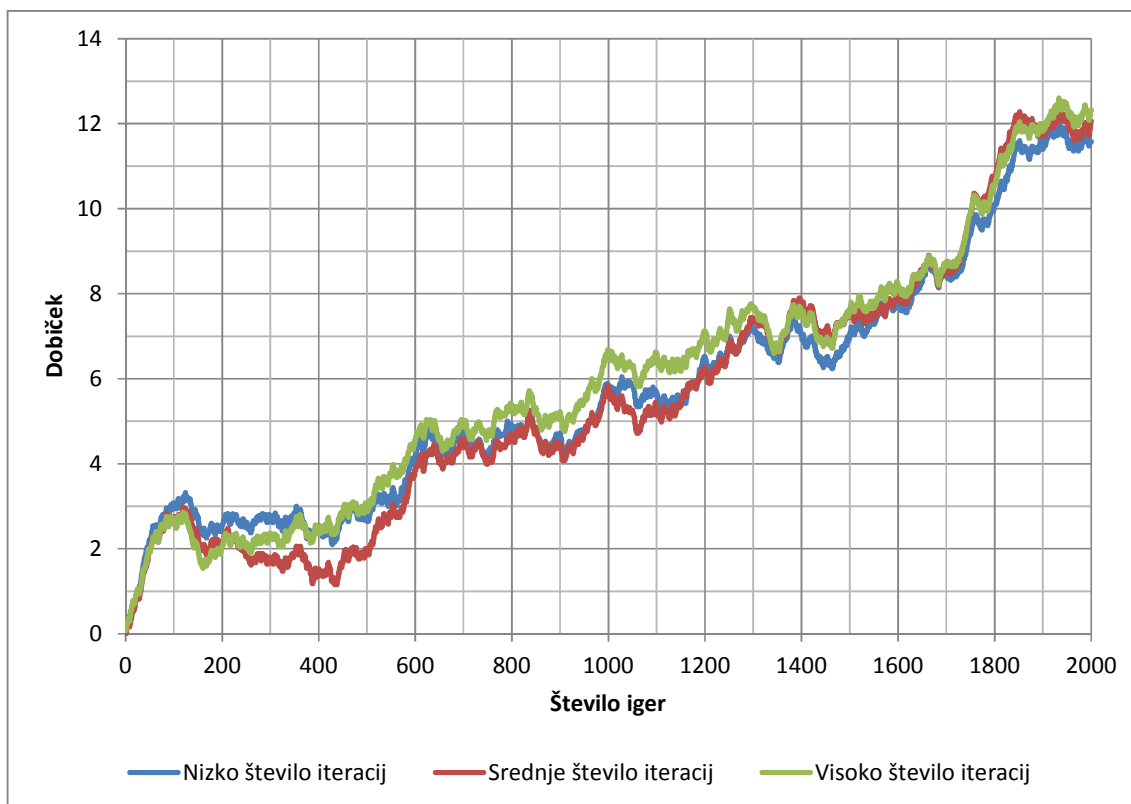
5.1.2 Ohranjanje drevesa

Med posameznimi simulacijami lahko drevo gradimo vedno znova, ali že naučeno ohranimo za naslednjo simulacijo. Rezultati simulacije s slednjim pristopom so razvidni v Tabeli 15.

Tabela 15: Prikaz učinkovitosti agenta MCTS z ohranjanjem drevesa proti agentu AlwaysCallBot.

Število iteracij algoritma MCTS	Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/veliko slepo stavo	Razlika povprečnega dobička/igro/veliko slepo stavo v primerjavi z osnovnim agentom MCTS
5000	11,58	0,0029	0,15	+0,1
10000	12,06	0,0030	0,15	+0,0
15000	12,32	0,0031	0,16	+0,0

Rezultati že na prvi pogled niso obetajoči. To gre pripisati predvsem veliki varianci v igri. Kar je v predhodnem krogu predstavljalo dobro potezo, lahko v naslednjem krogu pomeni tudi najslabšo. V primeru naključnega izbiranja to ne povzroči veliko škode, saj se akcije izbirajo naključno. Drugače pa je recimo pri načinu izbiranja UCT. Kot bomo videli v nadaljevanju (Tabela 16), lahko ohranjanje drevesa pripelje do vidnega poslabšanja rezultatov.



Slika 12: Graf poteka dobička agenta MCTS z ohranjanjem drevesa proti agentu AlwaysCallBot.

Tudi na Sliki 12 vidimo, da od osnovne simulacije ni veliko razlike v rasti dobička.

Za primer sem pognal simulacijo algoritma MCTS z ohranjanjem drevesa v kombinaciji z algoritmom UCT za izbiranje akcije. Rezultati so prikazani v Tabeli 16.

Tabela 16: Prikaz učinkovitosti agenta UCT z ohranjanjem drevesa proti agentu UCT brez ohranjanja drevesa.

Število iteracij algoritma MCTS	Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/veliko slepo stavo	Razlika povprečnega dobička/igro/veliko slepo stavo v primerjavi z UCT brez ohranjanja drevesa
10000	26,24	0,0065	0,33	-0,04

V kolikor primerjamo rezultate v Tabeli 16 z rezultatom izbiranja akcije UCT brez ohranjanja drevesa (Tabela 14), vidimo, da se rezultati simulacije zaradi ohranjanja drevesa dejansko nekoliko poslabšajo.

5.1.3 Model nasprotnika

V našem primeru je model nasprotnika zelo trivialen in intuitiven. Predvideva namreč samo, da nasprotnik ne bi odstopil, v kolikor mu to ne bi bilo potrebno. Z modelom nasprotnika popravimo tako algoritem izbire akcije kot algoritem simulacije.

- V algoritmu izbire akcije nasprotnik nikoli ne odstopi, če ima na razpolago možnost klicanja (nadaljevati igro, brez da bi vložil več denarja v igro).
- V algoritmu simulacije se ne simulira akcija odstopa, v kolikor je na razpolago akcija klicanja (nadaljevati igro, brez da bi vložil več denarja v igro).

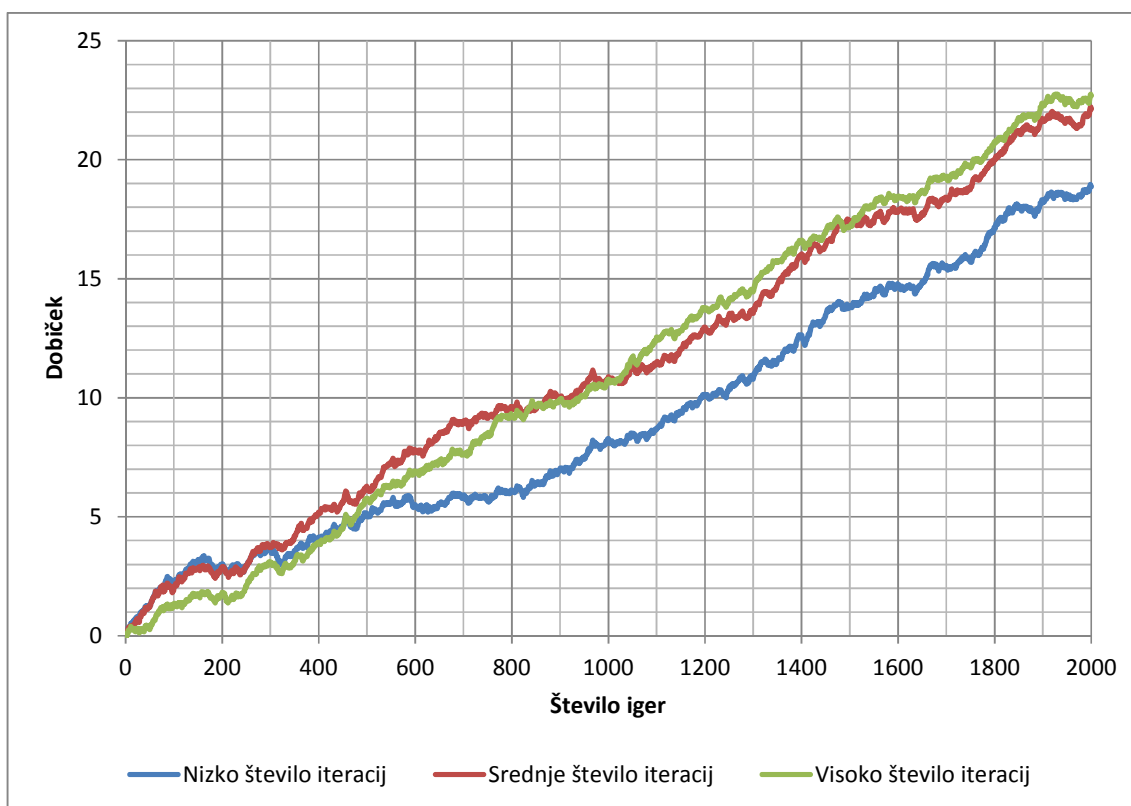
Rezultat implementacije modela nasprotnika je ta, da se večje število iteracij nameni v veje drevesa, ki so bolj obetavne za optimalno odločitev.

Rezultati aplikacije modela nasprotnika so razvidni v Tabeli 17.

Tabela 17: Prikaz učinkovitosti agenta MCTS z modelom nasprotnika proti agentu AlwaysCallBot.

Število iteracij algoritma MCTS	Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/veliko slepo stavo	Razlika povprečnega dobička/igro/veliko slepo stavo v primerjavi z osnovnim agentom MCTS
5000	18,86	0,0047	0,24	+0,10
10000	22,12	0,0055	0,28	+0,13
15000	22,69	0,0057	0,28	+0,12

Za model nasprotnika velja enako pravilo kot za ohranjanje drevesa. V kombinaciji z algoritmom UCT za izbiranje akcije se rezultati še bolj občutno izboljšajo.



Slika 13: Graf poteka dobička agenta MCTS z modelom nasprotnika proti agentu AlwaysCallBot.

Na Sliki 13 lahko vidimo, da so rezultati agenta MCTS z modelom nasprotnika, čeprav slabši, zelo podobni rezultatom agenta MCTS z algoritmom izbiranja akcije UCT – konstantna rast brez večjih padcev dobička.

5.1.4 Število simulacij na iteracijo algoritma MCTS

Na posamezno iteracijo algoritma se izvaja več simuliranih iger. Ker sem skušal ugotoviti vpliv števila simulacij igre na končni rezultat, sem jih variiral samo pri srednjem številu iteracij (10000).

Tabela 18: Prikaz učinkovitosti agenta MCTS z več simulacijami igre na iteracijo proti agentu AlwaysCallBot.

Število iteracij algoritma MCTS	Število simulacij na iteracijo	Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/veliko slepo stavo	Razlika povprečnega dobička/igro/veliko slepo stavo v primerjavi z osnovnim agentom MCTS
10000	10	12,40	0,0031	0,16	+0,01
10000	20	12,60	0,0032	0,16	+0,01
10000	50	12,98	0,0032	0,16	+0,01

Tabela 18 prikazuje, da se z večanjem števila simulacij na iteracijo končni dobiček počasi izboljšuje. Povečanje je pričakovano – več simulacij kot opravimo, bolj natančno dodelimo nagrado izbrani poti skozi graf v določeni iteraciji.

Za parameter torej velja enako pravilo kot za skupno število iteracij: več simulacij na iteracijo kot imamo, bolj optimalno bo deloval naš agent. Algoritem se z večanjem števila simulacij na iteracijo manj upočasni kot z večanjem števila iteracij, zato lahko parameter uporabimo za manjše izboljšave v končnem delovanju agenta.

5.2 Analiza igre proti različnim agentom

Za analizo igre proti različnim agentom sem se osredotočil na kombinacijo najbolj uspešnih parametrov iz prejšnjega poglavja. Uporabil bom samo srednje število iteracij (10000), ker iz rezultatov testov izhaja, da ni bistvene razlike pri končnem dobičku med srednjim številom in visokim številom iteracij. Optimizirane vrednosti simulacijskih parametrov se nahajajo v Tabeli 19.

Tabela 19: Optimizirane vrednosti parametrov.

Parameter simulacije	Vrednost
Število iteracij algoritma MCTS	10000
Število simulacij na iteracijo algoritma MCTS	50
Strategija izbiranja vozlišč	UCT izbira
Model nasprotnika	Trivialen model nasprotnika (brez odstopa)
Ohranjanje drevesa	Drevo se v vsaki simulaciji gradi od začetka

Ker se varianca pri igri proti bolj naprednim agentom občutno poveča, je za pridobitev zanesljivih podatkov potrebno izvesti več kot eno simulacijo. Izvedel sem 50 simulacij in izračunal povprečni rezultat.

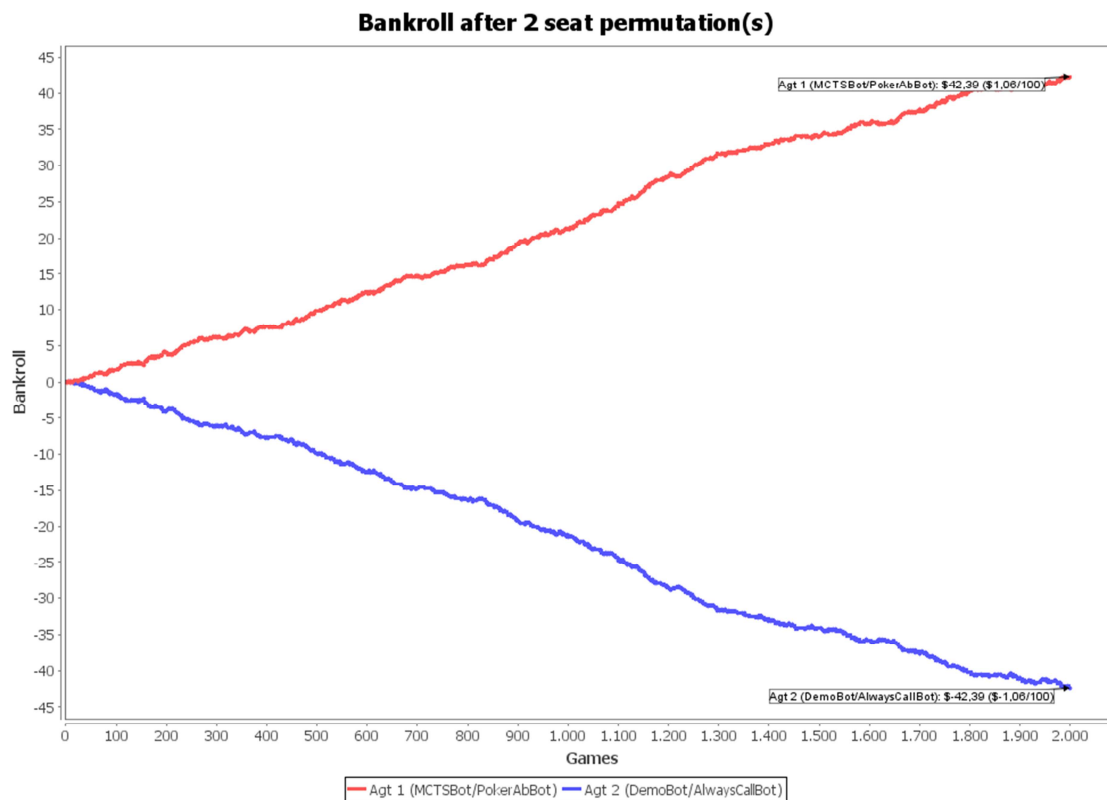
5.2.1 AlwaysCallBot

Proti testnemu nasprotniku sem preizkusil tudi optimalno konfiguracijo svojega agenta. Rezultati so prikazani v Tabeli 20.

Tabela 20: Prikaz učinkovitosti optimiziranega agenta MCTS proti agentu AlwaysCallBot.

Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/ veliko slepo stavo
42,39	0,0106	0,53

Višina končnega dobička je večja kot zgolj seštevek vseh izboljšav dobička posameznih parametrov. To gre pripisati dejstvu, da se optimizacije ne zgolj seštevajo, ampak imajo med seboj nelinearen vpliv.



Slika 14: Prikaz igre optimiziranega agenta MCTS proti agentu AlwaysCallBot.

Zelo dober rezultat sem dobil tudi kot posledico dejstva, da se preprost model nasprotnika zelo dobro prilega dejanskemu nasprotniku – agentu AlwaysCallBot. Lahko bi rekli, da je bila simulacija nekoliko nepoštena, ker sem lahko točno predvidel, kako se bo nasprotnik obnašal. Rezultat je torej lahko samo indikator, kako dobro se moj agent zaveda moči svoje roke. Potek dobička vidimo na Sliki 14.

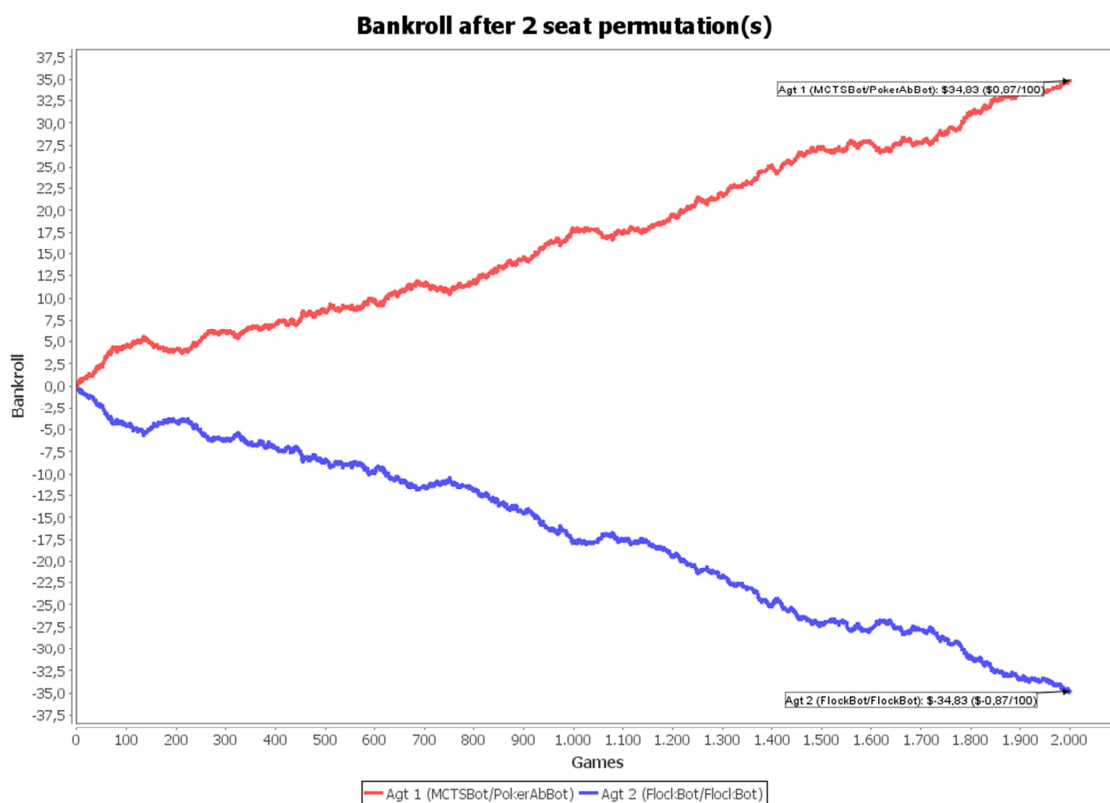
5.2.2 FlockBot

Nasprotnik igra v prvih treh krogih identično igro kot AlwaysCallBot, zato bistveno slabšega rezultata nisem pričakoval. Končni rezultati se nahajajo v Tabeli 21.

Tabela 21: Prikaz učinkovitosti optimiziranega agenta MCTS proti agentu FlockBot.

Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/ veliko slepo stavo
34,83	0,0087	0,44

Kljub majhni razliki pri samem delovanju je razlika v končnem rezultatu v primerjavi z agentom AlwaysCallBot zelo opazna. Na Sliki 15 lahko vidimo tudi, da je potek dobička nekoliko manj linearen prav zaradi različnih odločitev v končnici vsake igre.



Slika 15: Prikaz igre optimiziranega agenta MCTS proti agentu FlockBot.

S svojim obnašanjem agent FlockBot ne more bistveno vplivati na višino pota, na katerega ima dejansko vpliv samo v zadnjem krogu (zadnja četrtina igre). Nasprotniku (v tem primeru mojemu agentu MCTS) prepusti možnost določanja višine pota. Do razlike v primerjavi z agentom AlwaysCallBot pride samo zaradi višanj, s katerimi maksimizira svoj dobiček, in odstopov, s katerimi minimizira svojo izgubo, ki se zgodijo v zadnjem krogu.

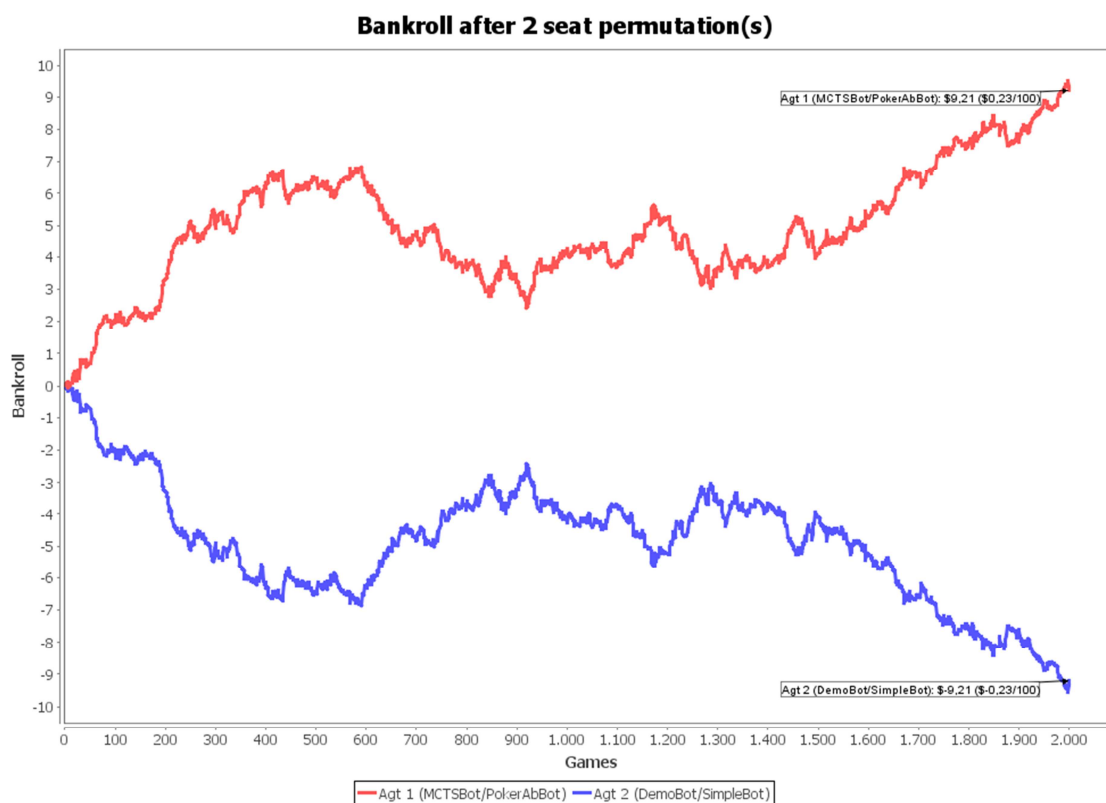
5.2.3 SimpleBot

SimpleBot je bil naš primarni nasprotnik, oziroma nasprotnik, katerega uspešnost smo poskušali vsaj izenačiti. Končni rezultat simulacij je prikazan v Tabeli 22.

Tabela 22: Prikaz učinkovitosti optimiziranega agenta MCTS proti agentu SimpleBot.

Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/ veliko slepo stavo
9,21	0,0023	0,12

Moj agent je, kot pričakovano na podlagi predhodnih rezultatov, premagal agent SimpleBot. V primerjavi z ostalimi rezultati lahko na Sliki 16 vidimo, da je potek pridobivanja dobička nelinearen (velik lokalni padec med igro 500 in 900).



Slika 16: Prikaz igre optimiziranega agenta MCTS proti agentu SimpleBot.

Nelinearnost lahko pripišemo večji varianci zaradi delno naključnega delovanja agenta. Do zdaj so bili vsi nasprotniki pretežno deterministični. SimpleBot uporablja naključje za določanje akcije glede na potencialno vrednost svoje roke. Poleg tega pa je tudi moj agent odvisen od verjetnosti. Ob vsaki akciji namreč preišče le majhen del celotne domene – pri temu mu pomaga drevo, ki to raziskovanje omejuje v najbolj obojavo smer.

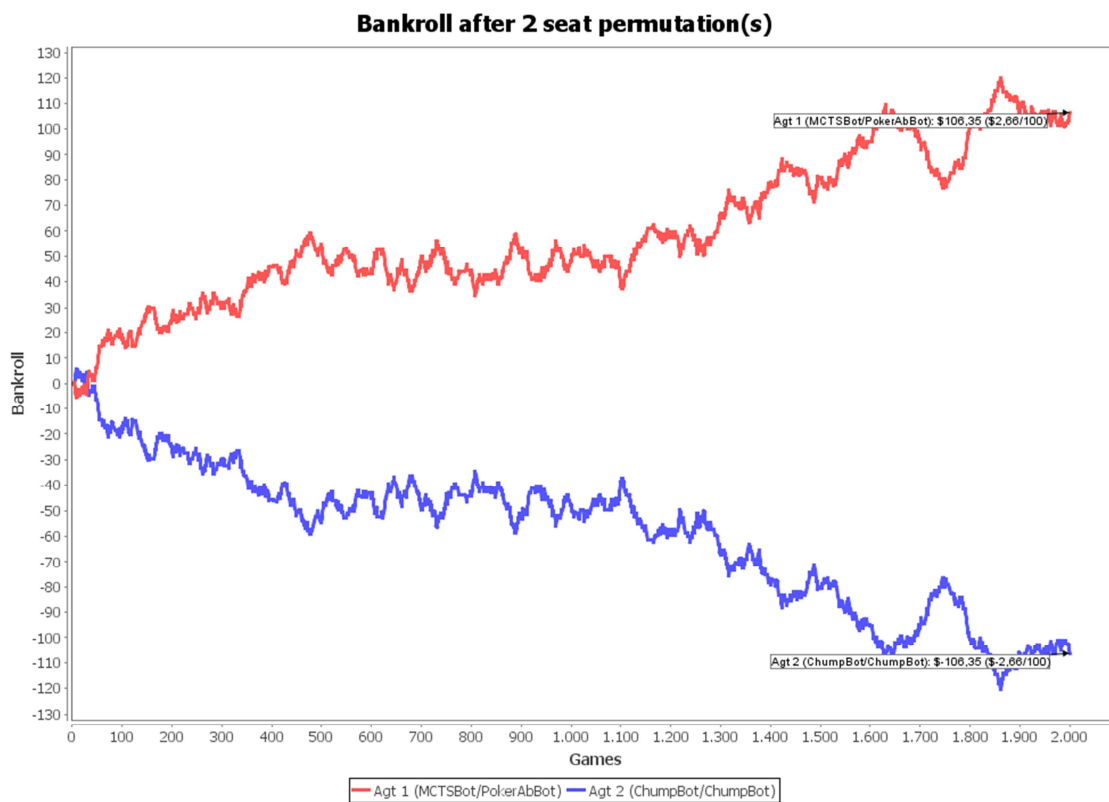
5.2.4 ChumpBot

ChumpBot je najbolj nepredvidljiv in naključju podvržen agent. Prav zaradi tega in zaradi svoje agresivnosti bi moral biti najtežji nasprotnik. Rezultati simulacij so razvidni v Tabeli 23.

Tabela 23: Prikaz učinkovitosti optimiziranega agenta MCTS proti agentu ChumpBot.

Končni dobiček	Povprečni dobiček/igro	Povprečni dobiček/igro/ veliko slepo stavo
106,35	0,0266	1,33

Rezultati simulacije so presenetljivi. Izkaže se, da je nasprotnikova igra preveč agresivna in posledično izgublja velike količine denarja. Končni dobiček mojega agenta je bil najvišji v vseh simulacijah. ChumpBot je izgubil več kot dvakrat več v primerjavi z najbolj enostavnim agentom AlwaysCallBot.

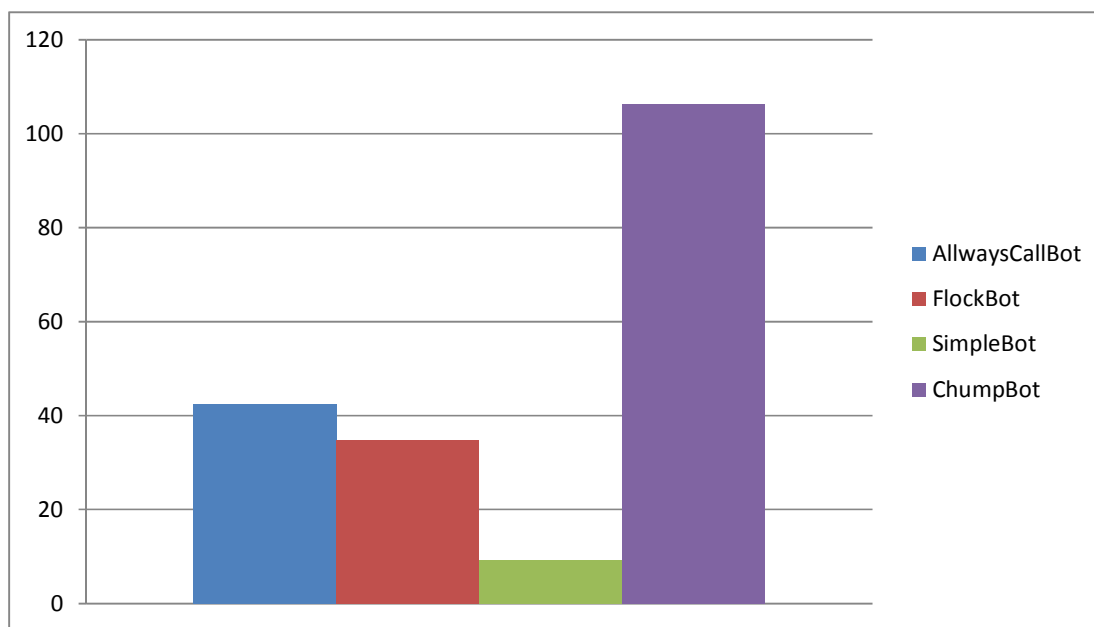


Slika 17: Prikaz igre optimiziranega agenta MCTS proti agentu ChumpBot.

Tudi iz grafa na Sliki 17 vidimo konstantno rast dobička skozi igre. Iz rezultatov lahko sklepamo, da naš agent pravilno ovrednoti svojo roko in odstopi iz igre, kadar je stava previsoka za trenutno roko. Nasprotnik tudi blefira. Ker pa moj agent nima dinamično prilagodljivega modela nasprotnika, ga nasprotnikovo blefiranje ne more zavesti v odstop iz igre, in je tako nasprotnikova igra manj kot optimalna.

5.3 Primerjava rezultatov igre proti različnim agentom

Ker so grafi poteka dobička v prejšnjem poglavju prikazani z različnimi skalami na Y osi (os dobička), bom na Sliki 18 grafično pokazal razmerje med končnimi dobički igre optimiziranega agenta MCTS proti različnim nasprotnikom.



Slika 18: Primerjava končnega dobička optimiziranega agenta MCTS proti različnim nasprotnikom.

Ob neposredni primerjavi končnega dobička optimiziranega agenta MCTS pri igri z različnimi agenti se lahko prepričamo o dejanski razliki v veščini igre izbranih nasprotnikov. Najbolj med vsemi izstopa agent ChumpBot, za katerega sem pričakoval, da ga bo najtežje premagati, vendar se je na koncu izkazalo ravno obratno. Tabelarni prikaz podatkov iz Slike 18 se nahaja v Tabeli 24.

Tabela 24: Primerjava rezultatov igre proti različnim agentom.

Ime nasprotnika	Končni dobiček mojega agenta
AllwaysCallBot	42,39
FlockBot	34,83
SimpleBot	9,21
ChumpBot	106,35

6 Zaključek

Implementacija algoritma je bila uspešno izvedena. Rezultati so pokazali, da je drevesno preiskovanje Monte Carlo primerna rešitev za učenje igralca pokra Texas Hold'em v načinu »heads up«. Število iteracij, ki so bile potrebne za doseg zadovoljivega nivoja igranja, je tudi dovolj nizko, da ga imamo možnost nadalje povečati in s tem še dodatno izboljšati rezultate. Videli smo, da je z vpeljavo bolj naprednega načina izbire akcije in simuliranja iger uspešnost agenta močno narasla, predvsem kot posledica bolj usmerjenega preiskovanja drevesa.

V primerjavi z determinističnimi in s psevdo determinističnimi metodami je algoritem MCTS uspešnejši predvsem zato, ker deterministične metode zaradi prevelikega števila možnih stanj igre niso natančne, ampak slonijo zgolj na približkih. Ker je osnova algoritma MCTS simuliranje iger, je njegova natančnost kritično odvisna od števila iteracij, ki jih algoritem opravi za vsako odločitev. Število iteracij, ki jih potrebujemo za zadovoljivo odločitev, z velikostjo domene eksponentno narašča. Posledično za posamezno odločitev potrebujemo tudi več časa.

Čas odločanja je ena šibkih točk algoritma MCTS. Druge metode z več domensko specifičnega znanja so hitreje zato, ker jim ni potrebno pregledovati celotne domene, temveč samo manjši del, ki je izpostavljen v določeni situaciji. Z vpeljavo več domensko specifičnega znanja v razne dele algoritma MCTS omejujemo del domene, ki jo raziskujemo za posamezno odločitev. Posledično potrebujemo manj iteracij za zadovoljivo odločitev, kar pripelje do hitrejšega odločanja. Govorimo lahko o razmerju med hitrostjo odločanja in količino domensko specifičnega znanja, ki ga vključimo v algoritem.

S pomočjo zbranih podatkov sem ugotovil, da:

- sta za igro Texas Hold'em pri algoritmu MCTS ključnega pomena algoritem izbire akcije in algoritem modela nasprotnika,
- MCTS za uspešno igro pokra ne potrebuje domensko specifičnega znanja,
- je MCTS že pri nizkem številu iteracij veliko boljši od determinističnih in psevdo determinističnih agentov,
- je MCTS v primerjavi z determinističnimi algoritmi zelo počasen, vendar še vedno dovolj hiter za normalno igro – primerljiv z odločitvenim časom človeka.

Algoritem blesti predvsem v velikih domenah z visokim številom stanj, kjer je analitična rešitev preobsežna, ali pa zaradi nepopolnega poznavanja domene nemogoča. Diplomsko delo je lahko osnova za nadaljnje raziskovanje algoritma MCTS, predvsem za igro poker Texas Hold'em. Z razvojem ogrodja za algoritem MCTS in določitvijo kritičnih parametrov simulacije za poker je postavljen temelj za aplikacijo na bolj kompleksne domene, kot je npr. igranje pokra No-limit Hold'em proti večjemu številu igralcev, kjer višina stav in večje število nasprotnikov močno zaostri problem iskanja optimalne odločitve.

7 Stvarno kazalo

Slika 1: Interakcija stroja z okoljem.	5
Slika 2: Prikaz preiskovalnega drevesa.	8
Slika 3: Prikaz drevesnega preiskovanja Monte Carlo.	9
Slika 4: Psevdo koda algoritma drevesnega preiskovanja Monte Carlo.	10
Slika 5: Program Proker Academy Pro.	22
Slika 6: Primer igre agenta, ki podpira programski vmesnik »Meerkat API«.	25
Slika 7: Grafični prikaz finančnega stanja agentov skozi igro.	26
Slika 8: Prikaz dobička determinističnega agenta, ki igra sam s seboj.	31
Slika 9: Prikaz dobička igre agenta SimpleBot proti agentu AlwaysCallBot.	34
Slika 10: Graf poteka dobička osnovnega agenta MCTS proti agentu AlwaysCallBot.	36
Slika 11: Graf poteka dobička agenta MCTS z izboljšanim algoritmom za izbiro akcije proti agentu AlwaysCallBot.	37
Slika 12: Graf poteka dobička agenta MCTS z ohranjanjem drevesa proti agentu AlwaysCallBot.	39
Slika 13: Graf poteka dobička agenta MCTS z modelom nasprotnika proti agentu AlwaysCallBot.	41
Slika 14: Prikaz igre optimiziranega agenta MCTS proti agentu AlwaysCallBot.	43
Slika 15: Prikaz igre optimiziranega agenta MCTS proti agentu FlockBot.	44
Slika 16: Prikaz igre optimiziranega agenta MCTS proti agentu SimpleBot.	45
Slika 17: Prikaz igre optimiziranega agenta MCTS proti agentu ChumpBot.	46
Slika 18: Primerjava končnega dobička optimiziranega agenta MCTS proti različnim nasprotnikom.	47
Tabela 1: Osnovni pojmi Texas Hold'em.	13
Tabela 2: Akcije pri Texas Hold'em.	14
Tabela 3: Krogi pri Texas Hold'em.	14
Tabela 4: Vrednotenje rok pri Hold'em.	16
Tabela 5: Število možnih rok nasprotnikov v stanju »pre-flop« v odvisnosti od števila igralcev.	17
Tabela 6: Prikaz večjega števila stanj na krog.	18
Tabela 7: Osnovni razredi pri programskem vmesniku »Meerkat API«.	23
Tabela 8: Osnovni dogodki pri agentu »Meerkat API«.	23
Tabela 9: Dodatni dogodki za agent »Meerkat API«.	24
Tabela 10: Parametri osnovne simulacije.	30
Tabela 11: Konfiguracije števila iteracij algoritma MCTS.	31
Tabela 12: Prikaz učinkovitosti agenta SimpleBot proti agentu AlwaysCallBot.	33
Tabela 13: Prikaz učinkovitosti osnovnega agenta MCTS proti agentu AlwaysCallBot.	35
Tabela 14: Prikaz učinkovitosti agenta MCTS z izboljšanim algoritmom za izbiro akcije proti agentu AlwaysCallBot.	37

Tabela 15: Prikaz učinkovitosti agenta MCTS z ohranjanjem drevesa proti agentu AlwaysCallBot.....	38
Tabela 16: Prikaz učinkovitosti agenta UCT z ohranjanjem drevesa proti agentu UCT brez ohranjanja drevesa.	39
Tabela 17: Prikaz učinkovitosti agenta MCTS z modelom nasprotnika proti agentu AlwaysCallBot.....	40
Tabela 18: Prikaz učinkovitosti agenta MCTS z več simulacijami igre na iteracijo proti agentu AlwaysCallBot.....	41
Tabela 19: Optimizirane vrednosti parametrov.	42
Tabela 20: Prikaz učinkovitosti optimiziranega agenta MCTS proti agentu AlwaysCallBot..	42
Tabela 21: Prikaz učinkovitosti optimiziranega agenta MCTS proti agentu FlockBot.	44
Tabela 22: Prikaz učinkovitosti optimiziranega agenta MCTS proti agentu SimpleBot.....	45
Tabela 23: Prikaz učinkovitosti optimiziranega agenta MCTS proti agentu ChumpBot.	46
Tabela 24: Primerjava rezultatov igre proti različnim agentom.	47

8 Viri in literatura

- [1] G. M. J.-B. Chaslot, *Monte-Carlo Tree Search*. 2010.
- [2] S. B. I. S. a. P. S. Guillaume Chaslot, "Monte-Carlo Tree Search: A New Framework for Game AI," 2010.
- [3] G. E. H. Gerritsen, *Combining Monte-Carlo Tree Search and Opponent Modelling in Poker*. 2010.
- [4] (2012, Dec.) Igre na srečo. [Online]. <http://www.igrenasreco.com/poker.html>
- [5] A. A. J. v. d. Kleij, *Monte Carlo Tree Search and Opponent Modeling through Texas Hold'em Poker*. 2012.
- [6] A. Kohne, *Monte-Carlo Tree Search in Chess Endgames*. 2012.
- [7] (2012, Dec.) Mathematica model. [Online]. http://en.wikipedia.org/wiki/Mathematical_model
- [8] (2012, Dec.) Monte Carlo method. [Online]. http://en.wikipedia.org/wiki/Monte_Carlo_method
- [9] (2013, Feb.) OpenTestBed. [Online]. <http://code.google.com/p/opentestbed/>
- [10] (2013, Feb.) PaBots. [Online]. <http://oursland.net/projects/pabots/>
- [11] (2013, Mar.) Poker probability. http://en.wikipedia.org/wiki/Poker_probability
- [12] (2012, Dec.) Texas hold 'em. [Online]. http://en.wikipedia.org/wiki/Texas_hold_%27em
- [13] (2013, May) PokerNews. [Online]. <http://si.pokernews.com/poker-izrazi/pot.htm>
- [14] G. G. G. C. Marc Ponsen, *Integrating Opponent Models with Monte-Carlo Tree Search in Poker*. 2010.
- [15] (2012, Dec.) Reinforced learning. [Online]. http://en.wikipedia.org/wiki/Reinforcement_learning
- [16] (2012, Dec.) Strojno učenje. [Online]. http://sl.wikipedia.org/wiki/Strojno_u%C4%8Denje