UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Žan Kafol

Preiskovanje pri igrah z nepopolno informacijo na primeru tršeta

MAGISTRSKO DELO

ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2014

UNIVERZA V LJUBLJANI

FAKULTETA ZA RAČUNALNIŠTVO IN INFORMATIKO

Žan Kafol

Preiskovanje pri igrah z nepopolno informacijo na primeru tršeta

MAGISTRSKO DELO

ŠTUDIJSKI PROGRAM DRUGE STOPNJE
RAČUNALNIŠTVO IN INFORMATIKA

Ljubljana, 2014

IZJAVA O AVTORSTVU MAGISTRSKEGA DELA

Spodaj podpisani Žan Kafol, z vpisno številko **63070040**, sem avtor magistrskega dela z naslovom:

Preiskovanje pri igrah z nepopolno informacijo na primeru tršeta

S svojim podpisom zagotavljam, da:

- sem magistrsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Marka Robnika-Šikonje,
- so elektronska oblika magistrskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko magistrskega dela,
- soglašam z javno objavo elektronske oblike magistrskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 20. septembra 2014 Podpis avtorja:

Zahvaljujem se prof. dr. Marku Robniku-Šikonji za vse strokovne nasvete, ideje in pomoč med pisanjem magistrske naloge. Iskrena zahvala gre tudi domačim in najbližjim za podporo med vsemi leti študija. Zahvalil bi se tudi vsem uporabnikom portala Briskula.si za temeljito testiranje in pripombe. Hvala tudi vsem ostalim, ki ste mi med študijem stali ob strani in me podpirali.

Vsem igralcem na Briskula.si.

Kazalo

Kazalo slik

Kazalo tabel

Seznam uporabljenih kratic

kratica	angleško	slovensko
AI	Artificial Intelligence	Umetna inteligenca
API	Application Programming Interface	Programski vmesnik
ARFF	Attribute-Relation File Format	Datoteka z relacijskimi atributi
BFS	Breadth-first search	Preiskovanje v širino
CSV	Comma Separated Value	Datoteka z vejicami ločenimi vrednostmi
DFS	Depth-first search	Preiskovanje v globino
ID	Iterative deepening	Iterativno poglabljanje
JAR	Java Archive	Paket java
JSON	JavaScript Object Notation	Notacija objektov JavaScript
k-NN	k-Nearest Neighbours algorithm	Metoda k-najbližjih sosedov
LSH	Locality-sensitive hashing	Krajevno zgoščevanje
MCTS	Monte Carlo Tree Search	Preiskovanje dreves Monte Carlo
PHP	PHP Hypertext Preprocessor	Programski jezik PHP
PIMC	Perfect Information Monte Carlo	Monte Carlo s popolno informacijo
PVS	Principal Variation search	Preiskovanje glavne inačice
TCP	Transmission Control Protocol	Protokol za nadzor prenosa
UCT	Upper Confidence Bounds Applied To Trees	Zgornja meja zaupanja uporabljena pri drevesih
XML	Extensible Markup Languate	Razširljiv označevalni jezik
α-β	Alpha-Beta pruning	Alfa-Beta rezanje
σ	Standard deviation	Standardni odklon

Povzetek

V magistrskem delu preizkušamo različne pristope za reševanje problema preiskovanja z nepopolno informacijo. Za primer smo izbrali igro s kartami tršet, kjer nepopolno informacijo predstavljajo karte v kupčku, to pa pomeni, da možne poteze igralcem niso vidne in na igro vpliva verjetnost. Glavni poudarek je na metodi preiskovanja dreves Monte Carlo (MCTS), ki temelji na naključnih simulacijah in preišče le del prostora. MCTS se je na tej domeni izkazal za uspešno metodo. Razvili smo prototip avtomatskega agenta za igranje igre, ga postopoma izboljševali s spreminjanjem parametrov ter vpeljevanjem hevristik ter merili njegovo uspešnost. V preiskovanje smo vključili tudi znanje, pridobljeno iz baze človeških iger ter testirali vpliv parametrov na uspešnost. Uspešnost smo ovrednotili na podlagi iger proti človeškim igralcem ter z medsebojnim igranjem različnih pristopov.

Ključne besede: Preiskovanje dreves Monte Carlo, nepopolna informacija, umetna inteligenca, tršet, ekspektiminimaks, preiskovanje

Abstract

We explore different approaches for solving problems with incomplete information. As an example a card game Tressette is chosen where the incomplete information is presented as cards still in the deck. This means that players cannot make deterministic strategies on possible outcomes or predict the moves of an opponent, because such moves are not guaranteed, but are possible with certain probability. The main emphasis is on the Monte Carlo tree search method (MCTS), which uses random sampling and simulates only a part of the search space. MCTS has proven to be a successful method in this domain. A prototype of an intelligent agent was developed for playing the game. The agent was gradually improved by tuning MCTS method parameters and by introducing new heuristics into the search. We used knowledge extracted from the database of human-played games in the agent to improve its efficiency. The agent was tested by different approaches playing against each other and against human players.

Keywords: Monte Carlo Tree Search, Incomplete information, Artificial Intelligence, Tressette, Expectiminimax, search algorithm, Tressette, game playing

Poglavje 1 **Uvod**

1.1 Motivacija in cilji

Preiskovanje je ključna tehnika pri reševanju številnih problemov. Z njo poskušamo reševati tudi mnoge igre, ki služijo kot poligon za razvoj novih idej in pristopov.

Na primeru igre s kartami tršet želimo raziskati problem preiskovanja v igrah z nepopolno informacijo, kjer je klasične tehnike preiskovanja potrebno dopolniti z verjetnostjo stanj in vzorčenjem prostora stanj.

Kot glavno metodo preiskovanja smo izbrali MCTS (angl. Monte Carlo Tree Search), ki je razmeroma nova metoda in se v literaturi pojavlja šele v zadnjih letih [1]. Za primer smo vzeli igro s kartami tršet, kjer nepopolno informacijo predstavljajo karte v kupčku.

K izbiri teme nas je motiviralo to, da inteligentnega agenta za igranje igre s kartami tršet še nismo zasledili.

Na portalu za igranje igre s kartami Briskula.si smo uspešnost izdelane rešitve testirali proti človeškim igralcem. Naš cilj je bil izdelati rešitev, ki lahko konkurira povprečnemu igralcu.

1.2 Problemsko ozadje

V preteklosti se je večina raziskav umetne inteligence v igrah osredotočala na igre s popolno informacijo (kot na primer šah ali dama). Popolna informacija v igrah pomeni, da je stanje igre vidno vsem igralcem ob vsakem času. Na ta način lahko načeloma preiščemo vsa možna stanja v igri, na primer z drevesnim preiskovanjem – preiskovanjem v globino (angl. Depth-first search) ali v širino (angl. Breadth-first search). Pri igrah z nepopolno informacijo je stanje igre le deloma znano [2].

Z nepopolno informacijo se soočajo tako ljudje kot agenti. Za igre z nepopolno informacijo je izdelava dobrega računalniškega agenta računsko zelo zahtevna, število možnih različnih stanj preseže prostorske in računske zmogljivosti. Igre z nepopolno informacijo običajno vsebujejo

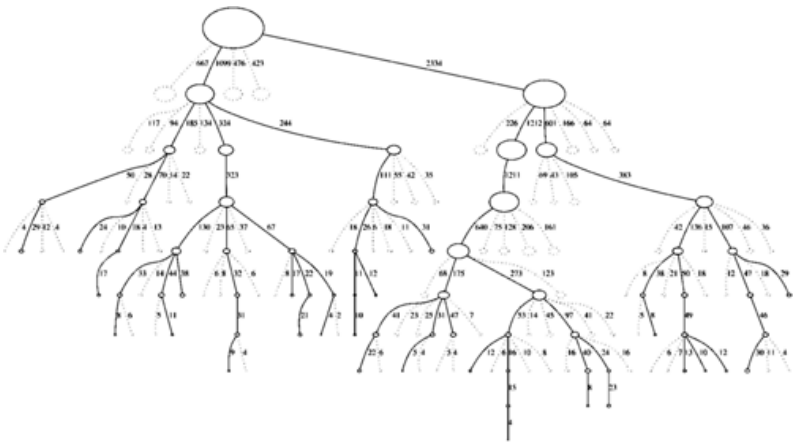
kombinacije kompleksnih nalog, kot na primer hevristično preiskovanje, rekonstrukcijo verjetnostnih stanj (angl. belief state reconstruction) in modeliranje nasprotnika [3].

Priljubljen način reševanja problema nepopolne informacije je, da se problemu izognemo. Namesto preiskovanja vseh kombinacij igre vzorčimo različice igre s popolno informacijo ter jih ovrednotimo (eksaktno ali hevristično). Ta pristop, imenovan Perfect Information Monte Carlo (PIMC), je dosegel odlične rezultate pri igrah s kartami, kot na primer pri bridžu, skatu in srcih [4].

Za izdelavo dobrega računalniškega agenta poskušamo kombinirati preizkušene tehnike, kar pomeni vpeljevanje hevristik, preiskovanje s popolno informacijo in upoštevanje verjetnosti stanj.

1.2.1 Osnove preiskovanja

Preiskovanje je tehnika reševanja problemov, pri kateri se gradijo možna stanja, rešitev pa se poišče med njimi. Zbirka možnih stanj je pri igrah navadno predstavljena kot drevo. Za primer si poglejmo sliko 1, kjer je kot drevesna struktura predstavljeno preiskovalno drevo pri metodi Monte Carlo Tree Search. Vsako vozlišče v drevesu predstavlja stanje igre, vsaka veja v drevesu pa predstavlja eno potezo.



Slika 1: Iskalno drevo pri metodi MCTS [5].

Za preiskovanje drevesnih struktur obstaja več algoritmov. Najbolj poznani in uporabljeni so preiskovanje v globino (DFS), preiskovanje v širino (BFS) in iterativno poglabljanje (ID). Algoritem DFS se z vsakim korakom pomika v naslednike vozlišč. BFS preišče vsa vozlišča po nivojih. Iterativno poglabljanje uporablja tehniko poglabljanja, vsaka iteracija pa ima omejeno globino preiskovanja, ki se postopoma povečuje. Funkcionalno je ID podoben BFS, le da je prostorsko manj zahteven. Našteti algoritmi so primerni tudi za igre z enim igralcem,

na primer iskanje poti iz labirinta, reševanje Rubikove kocke in podobno. Pri takšnih igrah vsako vozlišče predstavlja eno stanje v igri (na primer konfiguracija kocke), ena veja v drevesu pa predstavlja eno potezo ali premik. Rešitev je samo ena.

Pri igrah za dva ali več igralcev se običajno preiskovanje v globino spremeni tako, da se kot en premik upoštevata dve potezi, ena od agenta in druga od nasprotnika. Igra je lahko konkurenčna ali sodelovalna. Igralci bodo izbirali strategije tako, da bo njihov dobiček maksimalen. Pri konkurenčnih igrah igralci želijo maksimizirati le svoj dobiček. Poleg tega igralci upoštevajo vse informacije o igri, ki so jim na voljo, tako da lahko predvidevajo tudi poteze drugih igralcev [6]. Idealna rešitev, vsaj za igre z dvema igralcema, je uporabiti tehniko, ki poišče Nashevo ravnovesje. Nashevo ravnovesje v igri je vzpostavljeno, ko nobeden od igralcev ne želi spremeniti svoje strategije kljub popolni informaciji o nasprotnikovih možnih potezah. To zagotavlja idealno igro proti idealnemu nasprotniku. Takšen pristop je zaradi velikega problemskega prostora računsko prezahteven, razen za najenostavnejše igre. Ko nobeden od igralcev ne želi odstopiti od svoje strategije kljub vsem informacijam, ki so na voljo, je vzpostavljeno Nashevo ravnovesje. Koncept je odkril John Nash z definicijo:

Naj bo (S, f) igra z n igralci, kjer je S_i množica strategij za igralca i, $S = S_1 \times S_2 \times \ldots \times S_n$ je množica vseh strategij vseh igralcev in $f = (f_1(x), \ldots, f_n(x))$ je funkcija dobičkov za $x \in S$. Naj bo x_i strategija igralca i in x_{-i} strategija vseh igralcev, razen igralca i. Ko vsak igralec $i \in \{1, \ldots, n\}$ izbere strategijo x_i, kar predstavlja strategije $x = (x_1, \ldots, x_n)$, potem igralec i prejme dobiček $f_i(x)$. Dobiček je odvisen od vseh izbranih strategij vseh igralcev. Strategije $x^* \in S$ predstavljajo Nashevo ravnovesje, ko nobeno odstopanje od katere koli izbrane strategije katerega koli igralca ne prinese večjega dobička, torej

$$\forall i, x_i \in S_i : f_i(x_i^*, x_{-i}^*) \geq f_i(x_i, x_{-i}^*)$$

Nashevo ravnovesje je uporabljeno v več disciplinah, od vedenjske ekologije do ekonomije. Če želimo preizkusiti Nashevo ravnovesje, moramo pogledati vse strategije vseh igralcev. Nashevo ravnovesje obstaja, če noben igralec ne spremeni svoje strategije kljub poznavanju akcij svojih nasprotnikov. Za primer si poglejmo enostavno igro dveh igralcev, kjer vsak igralec lahko izbere potezo A in prejme dobiček 1€ ali potezo B in izgubi 1€. Možni izidi so predstavljeni v sliki 2. Pri izidu A,A sta oba igralca imela dobiček 1€, pri izidu B,B pa sta oba igralca izgubila 1€.

2. igralec \ 1. igralec	A	B
A	1, 1	1, -1
B	-1, 1	-1, -1

Slika 2: Prikaz možnih izidov trivialne igre za ilustracijo Nashevega ravnovesja.

Oba igralca bosta izbrala strategijo A in imela dobiček 1€. Če se izbira posameznega igralca razkrije nasprotniku, igralca kljub temu ne bosta spremenila svoje strategije. Vedenje o nasprotnikovi potezi ne spremeni vedenja igralcev. Izid A,A torej predstavlja Nashevo ravnovesje [7].

V teoriji iger se pogosto omenja lastnost ničelne vsote (angl. zero-sum game). Igra ničelne vsote je matematična predstavitev situacije, ko je igralčev dobiček (ali izguba) natančno uravnotežena z nasprotnikovo izgubo (ali dobičkom). Če se vsi dobički vseh igralcev v igri seštejejo, vse izgube igralcev v igri pa odštejejo, bo vsota natančno nič. Igre s kartami so običajno igre z ničelno vsoto, saj vsaka pobrana karta pomeni za nasprotnika izgubo. Na drugi strani neničelna vsota opisuje situacijo, kjer vsota dobičkov in izgub nanese na več ali manj od nič. Igre z ničelno vsoto so strogo konkurenčne, medtem ko so lahko igre z neničelno vsoto konkurenčne ali sodelovalne.

V teoriji iger problem Nashevega ravnovesja in lastnost ničelne vsote rešuje algoritem minimaks, ki preiskuje maksimalni dobiček in minimalno izgubo v igri. Minimaks si lahko predstavljamo kot algoritem, ki poskuša *maks*imizirati *min*imalen dobiček. Minimaks deluje pri igrah z n igralci in popolno informacijo. V vsakem koraku rekurzije upošteva, ali je na potezi maksimirajoči igralec ali minimizirajoči igralec. Variacija algoritma minimaks je ekspektiminimaks, kjer na igro vplivajo tudi elementi naključnosti, kot na primer met kocke. Deluje na enak način kot minimaks, le da izide upošteva z določeno verjetnostjo. Običajno negotove poteze v ekspektiminmaksu povprečimo. Performančno se minimaks in njegove variacije izboljšajo z metodo alfa-beta rezanja (αβ), kjer se preiskovanje ustavi v vozliščih, ki so slabša od že pregledanih [8].

Drug postopek za preiskovanje z nepopolno informacijo, ki smo ga podrobneje pregledali, je MCTS, izpeljan iz metode Monte Carlo. Metoda Monte Carlo temelji na večkratnem naključnem vzorčenju. MCTS simulira naključne igre in je iterativen postopek sestavljen iz štirih korakov: izbira (angl. selection), razširitev (angl. expansion), simulacija (angl. simulation) in vračanje informacij (angl. backpropagation). Ti štirje koraki se ponavljajo,

dokler je čas še na voljo. MCTS ni specifičen algoritem, ampak metoda, saj ne zapoveduje neposrednih pravil za katerega koli od štirih naštetih korakov. MCTS ne določa izbire vozlišča, iz katerega se bo vršila simulacija, ali kako naj se simulacija vrši in kako naj se rezultati simulacije vračajo nazaj po drevesu [3]. Pri izbiri vozlišča se pojavi problem eksploatacije (koliko časa namenimo preiskovanju obetavnih vozlišč) in eksploracije (koliko časa namenimo preiskovanju novih vozlišč), ki ga učinkovito rešuje algoritem UCT (Upper Confidence Bounds applied to Trees) [3].

Pri izdelavi prototipa agenta za igranje igre s kartami tršet uporabljamo metodo MCTS dokler imamo nepopolno informacijo, igro pa zaključimo s preiskovanjem z algoritmom minimaks.

1.2.2 Zgradba dela

V drugem poglavju bomo opisali igro tršet, pravila igre, hevristične strategije ter preiskovalni prostor. V tretjem poglavju bomo podrobneje razložili preiskovanje v igrah, način reševanja problemov, obstoječe rešitve ter sorodne raziskave. V četrtem poglavju predstavimo in opišemo naše rešitve, pristope in postopke izboljšav. V petem poglavju sledi evalvacija izdelane rešitve. Uspešnost rešitve smo izmerili z igranjem proti različnim pristopom ter z igranjem proti človeškim igralcem. Rešitev smo izdelali v programskem jeziku java, s katerim smo tudi testirali različne pristope. Testiranje proti človeškim igralcem je potekalo na spletni aplikaciji Briskula.si, napisani v programskem jeziku PHP za strežniški del in programsko logiko ter v orodjih HTML, JavaScript, CSS in Flash (ActionScript 2) za uporabniški del aplikacije. Podatki se hranijo v sistemu za upravljanje relacijskih podatkovnih baz MySQL.

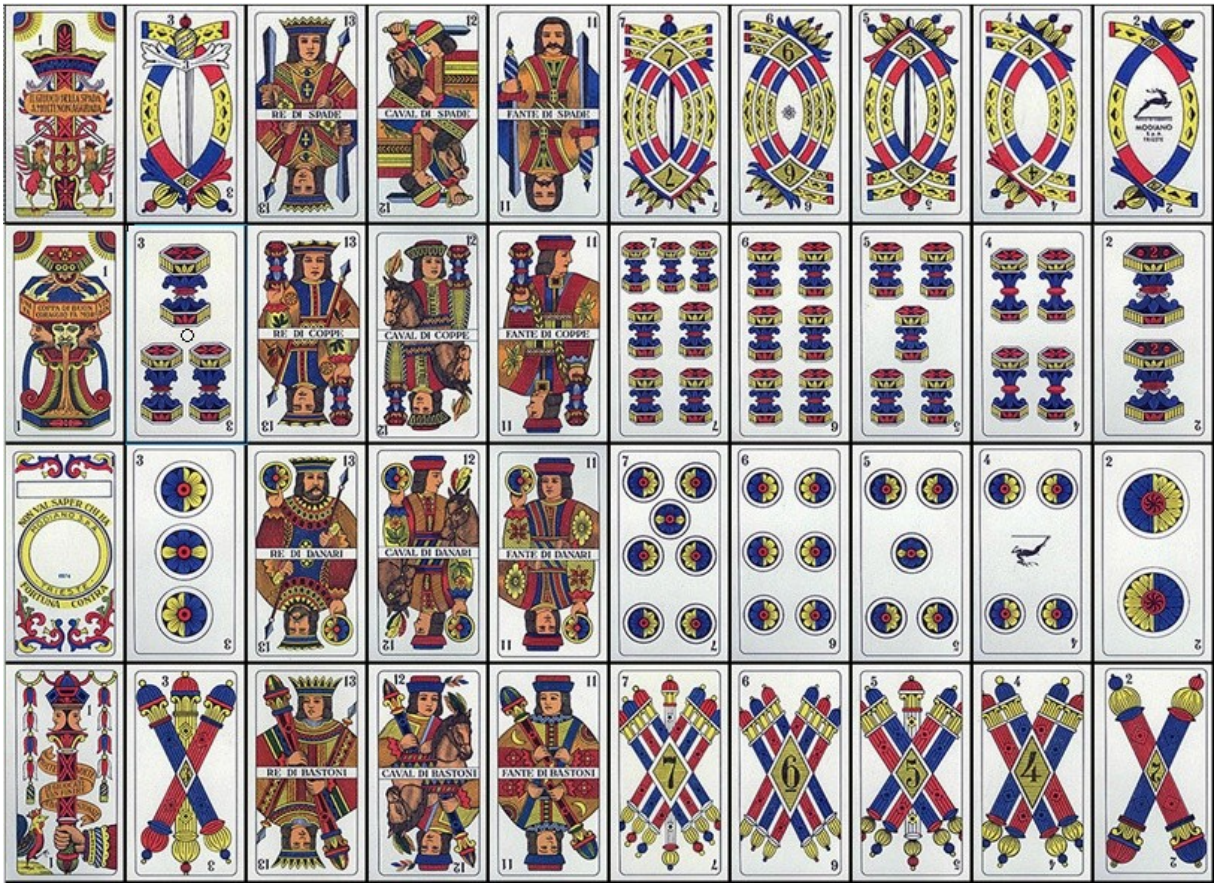
V zaključku povzamemo narejeno in izpeljemo glavne ugotovitve. Opišemo prednosti in slabosti ter kaj bi lahko naredili drugače oziroma možnosti za izboljšave.

Poglavje 2 **Opis tršeta**

Tršet je ena izmed najbolj igranih iger s kartami v Italiji, mnogi jo poznajo tudi v Slovenskem primorju, veliko jo igrajo tudi na Hrvaškem, ob obali Jadranskega morja in še posebej v Dalmaciji. V tem poglavju razložimo pravila igre ter njen preiskovalni prostor.

2.1 Pravila igre tršet

Za igro tršet je namenjen set štiridesetih kart, imenovan italijanski oziroma sicilijanski / scopa komplet kart, prikazan na sliki 3.



Slika 3: Igralne karte za tršet od leve proti desni: as, 3, kralj, kaval, fant, 7, 6, 5, 4 in 2 ter skupine sablje, pokali, denar in palice.

Karte v kompletu so razdeljene na štiri barve oziroma skupine po 10 kart. Skupine so pokali (it. Coppe), sablje (it. Spade), palice (it. Bastoni) in denar (it. Danari). Vsaka skupina kart vsebuje kralja, konjenika (kaval), fanta, as ter karte s številkami 7, 6, 5, 4, 3 in 2. S tem kompletom se lahko igra več iger, najbolj priljubljene so briškola (angl., it. briscola, hr. briškula, šp. brisca), tršet (angl., it. tressette, hr. trešeta) in škopa (angl., it. scopa).

Različne države imajo svoje različice igre, ki se razlikujejo v malenkostih, glavna pravila pa ostajajo enaka. Tršet se lahko igra z dvema, tremi ali štirimi igralci. V igri s štirimi igralci igrata dve nasprotni ekipi po dva igralca. Karte se pred začetkom igre premešajo in razdelijo med igralce, tako da ima vsak igralec na začetku v roki 10 kart. To pomeni, da pri igri dveh ali treh igralcev ostane nekaj kart še v kupčku, medtem ko se pri igri štirih igralcev razdelijo vse karte. Karte v igralčevih rokah so drugim skrite, informacijo o tem, katere karte ima igralec v roki, je prepovedano deliti z ostalimi, to velja tudi za igro štirih igralcev. Kupček preostalih kart je obrnjen navzdol, tako da so karte skrite. Igra se začne tako, da igralec, ki je na potezi, izbere eno karto in jo vrže na mizo. Običajno se igra v smeri urinega kazalca. Odvrženo karto na mizi vidijo vsi igralci. Igralec, ki je prvi na potezi, lahko začne s katerokoli karto, ki jo ima v roki, igralci, ki mu sledijo, pa morajo izbrati karto iz iste skupine (odgovarjanje na barvo). V primeru, da igralec nima nobene karte iz iste skupine, kot je prva karta na mizi, lahko odvrže katero koli karto, a s tem ne more pobrati vložka. Ko vsi igralci za mizo odvržejo po eno karto, je konec runde in določi se zmagovalca runde. Zmagovalna karta je najmočnejša na mizi iz skupine prve padle karte. Zmagovalec runde (oziroma zmagovalna ekipa) pobere vse karte na mizi in jih pospravi. Če so karte še v kupčku (torej pri igri dveh ali treh) na koncu runde vsak igralec iz kupčka vzame še po eno karto, začenši z zmagovalcem runde, tako da ima vsak v roki ponovno 10 kart. Karto prejeto iz kupčka mora vsak igralec pokazati nasprotnikom, tako da lahko ta delček informacije pri igralnem agentu uporabimo (na primer za to karto upoštevamo večjo verjetnost). Naslednjo rundo začne zmagovalec prejšnje runde in igra se ponavlja, dokler ne zmanjka vseh kart. Na koncu igre vsak igralec oziroma ekipa prešteje točke pobranih kart in zmagovalec igre je igralec ali ekipa z največ točkami.

Pri igri za štiri igralce sta igralca v isti ekipi za mizo postavljena diagonalno – sedita drug nasproti drugega in na potezi je izmenoma po en igralec iz vsake ekipe. Pri igri v treh se ena izmed kart izloči iz igre (običajno karta z najmanj točkami in močjo, na primer štirica), zato da je v igri 39 kart in se jemanje izide.

Točkovanje in moč kart pri tršetu sta različni. Karta z veliko točkami ni nujno najmočnejša. Trojka je najmočnejša karta v igri in pobere vse ostale karte v svoji skupini. Po moči trojki sledi dvojka, nato as, kralj, kaval, fant, 7, 6, 5 in 4. As ima največ točk – eno točko – medtem

ko imajo trojka, dvojka, kralj, kaval in fant po ⅓ točke. Karte 7, 6, 5 in 4 so prazne, torej nimajo točk, imenujejo se tudi »liše«. Ena izmed običajnih hevristik v igri je torej strategija, kako pobrati asa, ki ni najmočnejša karta v igri. Ker je potrebno odgovarjati na barvo, lahko tudi šibka karta pobere rundo, če je bila odvržena prva in v rundi ni padla močnejša karta iz te skupine. Igralec, ki pobere zadnjo rundo, dobi še eno dodatno točko, kar nanese skupaj 11⅔ točke, igralci pa na koncu igre točke zaokrožijo navzdol in si pišejo 11 točk, če poberejo vse karte in zadnjo rundo. V našem prototipu in strežniškem delu aplikacije smo zaradi poenostavitve in v izogib delu s tretjinami vse točke pomnožili s 3, tako da je skupno število točk v igri 35. As in zadnja pobrana runda prineseta 3 točke, dvojka, trojka, kralj, kaval in fant pa imajo po eno točko.

Tršet je konkurenčna igra, kjer vsak igralec želi izbrati takšno strategijo, da bo njegov dobiček maksimalen. Variacija igre tršet je kifameno (Ciapanò ali Chi fa meno) – »kdor ima manj« – kjer je cilj igre zbrati čim manj točk. Pravila igre so enaka kot pri tršetu, vključno z močjo in točkovanjem kart ter zadnjo potezo, le da je zmagovalec igre tisti igralec ali ekipa, ki zbere manj točk. Strategija igre je zato popolnoma drugačna.

Na izid igre vpliva mnogo naključnih dejavnikov. Slaba začetna konfiguracija kart tudi najboljšemu igralcu ne bo prinesla zmage, zato smo morali pri testiranju uspešnosti to upoštevati. Ker je varianca v igri za štiri igralce še večja, smo se pri našem testiranju in prototipu omejili na igro z dvema igralcema.

2.2 Preiskovalni prostor

V igri za dva in tri nekaj kart ostane v kupčku. To predstavlja nepopolno informacijo, saj možnih igralčevih potez ni mogoče z gotovostjo napovedati, potrebno je upoštevati verjetnost, s katero ima igralec določene karte. To imenujemo verjetnost stanja.

Ker igralci vsako rundo pokažejo karto, ki jo vzamejo iz kupčka, postopoma dobivamo vedno več informacij o igri. Od desete poteze naprej, ko kart ni več v kupčku, imamo popolno informacijo.

V kompletu je 40 kart, na začetku ima vsak od igralcev v roki 10 kart, torej 20 kart ostane v kupčku. To narekuje $C = \binom{40}{10} = \frac{40!}{10! \times (40-10)!} = 847.660.528$ začetnih konfiguracij kart za enega igralca oziroma $\binom{40}{20} = 137.846.528.820$ začetnih konfiguracij kart za oba igralca. C označuje število možnih začetnih konfiguracij nasprotnika, ki jih ne poznamo zaradi nepopolne informacije in jih moramo upoštevati z določeno verjetnostjo. Iz začetne

konfiguracije lahko igralec na potezi izbere poljubno veljavno karto, kar še poveča število možnih izidov. Razvidno je, da je simulacija vseh kombinacij z upoštevanjem verjetnosti računsko prezahtevna za igro v realnem času.

Za izgradnjo uspešnega igralnega agenta moramo preiskati čim večji del preiskovalnega prostora in ga opremiti z najbolj obetavnimi strategijami ter s hevristikami pridobljenimi iz baze človekovih iger. Klasične in enostavne hevristične strategije pri tršetu so prisiliti nasprotnika, da odvrže karto z veliko točkami, ki jo lahko poberemo. To lahko dosežemo tako, da med igro zaporedoma odmetavamo močne karte tiste barve, ki je imamo veliko. Ker mora nasprotnik odgovarjati na barvo, bo moral sčasoma odvreči tudi ase, kralje, kavale ali fante. Naivna strategija je, da se močne karte odvržejo šele na koncu, saj lahko takrat z večjo gotovostjo predvidimo poteze nasprotnika, lažje pa je tudi izbrati poteze, ki poberejo zadnjo rundo, ki prinese dodatno točko. Med hevristike lahko umestimo tudi izkoriščanje priložnosti za pobiranje velike količine točk kadarkoli v igri. Za primer lahko vzamemo, da je nasprotnik prvi na potezi in vrže kralja. V kolikor imamo asa iste barve, ga vržemo in na ta način poberemo kar 4 točke, kar je že več kot petina točk potrebnih za zmago. Paziti je potrebno na to, ali je morda takšna poteza žrtvovanje nasprotnika za kasnejšo boljšo strategijo, torej ali nam je nasprotnik nastavil vabo.

Poglavje 3 **Preiskovanje v igrah**

V tem poglavju opišemo različne metode preiskovanja pri igrah, razložimo popolno in nepopolno informacijo ter obravnavamo sorodne raziskave.

3.1 Sorodne raziskave

Pri preiskovanju z nepopolno informacijo so se metode MCTS v zadnjih letih izkazale pri igrah, še posebej algoritem UCT [1]. Metode Monte Carlo so primerne, ko običajne minimaks tehnike ne dajejo dobrih rezultatov zaradi velikosti iskalnega prostora ter zaradi težavne priprave hevristične evalvacijske funkcije, ki ocenijo stanje, če izčrpno preiskovanje ne pride do končnega vozlišča. O uspešnosti metode Monte Carlo poročajo, na primer v članku Monte Carlo tree search in Kriegspiel (2010) [3], kjer so si metodo izbrali zaradi uspešnosti pri igri Phantom Go (variacija igre Go z nepopolno informacijo). Kriegspiel je variacija šaha z nepopolno informacijo. V družini MCTS algoritmov je priljubljen UCT [5], saj ima dobro razmerje med preiskovanjem novih in že poznanih uspešnih poti (angl. explore or exploit) [3]. UCT je omenjen praktično v vsaki raziskavi metod Monte Carlo.

MCTS je bil uspešno uporabljen na igri s kartami Magic: The Gathering [1], v diplomskih delih pa je bil uporabljen tudi za tarok [9] in poker [10]. Tršet in tarok sta dovolj različna, da obstoječi robotski agenti za tarok pri tršetu ne pridejo v poštev, hkrati pa sta v nekaterih delih igre dovolj podobna, da smo si lahko pomagali z literaturo, predvsem pri izbiri metode preiskovanja in njenih parametrov.

Pri iskanju obstoječih agentov za tršet smo odkrili odprtokodni projekt Tutegame [11], ki vsebuje robotskega agenta za špansko igro tute. Tute je podoben tršetu, za preiskovanje pa so avtorji uporabili ekspektiminimaks, ki izvaja izčrpno preiskovanje s hevristično evalvacijo stanj, ko pri preiskovanju zmanjka časa.

3.2 Neinformirano preiskovanje

Pri igrah za enega igralca rešitev običajno iščemo z osnovnimi algoritmi neinformiranega preiskovanja. Za lažje razumevanje delovanja algoritma minimaks in njegovih variacij najprej razložimo delovanje treh osnovnih preiskovalnih algoritmov na drevesih.

Graf je množica točk (vozlišč) in povezav med njimi. Drevo je acikličen povezan graf. Vsako vozlišče z izjemo korena ima natanko eno nadrejeno vozlišče (angl. parent node) in nič ali več podrejenih vozlišč (angl. child node). Vozlišče brez podrejenih vozlišč imenujemo list (angl. leaf node). Vsako vozlišče predstavlja eno stanje igre, povezava med dvema vozliščema pa potezo. Problem je definiran z enim ali več začetnimi stanji, enim ali več končnimi stanji ter s prehodi med stanji. Algoritem, ki preiskuje drevo, začne preiskovanje iz korena drevesa, nato pa se postopoma poglablja v naslednike oziroma v sosednja vozlišča. Rešitev problema je pot od začetnega do končnega stanja. Optimizacijske probleme predstavimo tako, da povezavam v grafu dodamo cene. Cena rešitve je po navadi vsota vseh povezav vzdolž rešitvene poti. Različni vrstni redi preiskovanja imajo različne prostorske in časovne zahteve.

3.2.1 Preiskovanje v globino

Preiskovanje v globino (angl. depth-first search ali DFS) se začne v korenu drevesa, nato pa med nasledniki izbere prvega. DFS se dobro prilega rekurziji, a ne zagotavlja, da je najdena rešitev tudi najkrajša. Časovna zahtevnost algoritma DFS je $O(b^m)$, prostorska zahtevnost pa $O(bm)$, kjer je b faktor vejanja grafa in m maksimalna globina prostora stanj.

Variacija algoritma DFS je depth-limited search (DLS), ki je DFS z omejeno globino. Ko algoritem DLS doseže določeno globino, se iskanje pri tem vozlišču ustavi in začne se vračanje.

3.2.2 Preiskovanje v širino

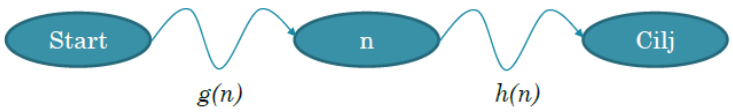
Algoritem preiskovanja v širino (angl. breadth-first search ali BFS) med alternativnimi vozlišči izbere tisto, ki je najbližje začetnemu stanju. BFS vedno najprej najde najkrajšo pot. Ker je potrebno voditi evidenco sosednjih vozlišč, je prostorska zahtevnost algoritma mnogo večja od DFS in je eksponentna, medtem ko ima DFS linearno prostorsko zahtevnost. Časovna in prostorska zahtevnost BFS je $O(b^{d+1})$.

3.2.3 *Iterativno poglabljanje*

Iterativno poglabljanje (angl. iterative deepening depth first search ali ID) je implementirano z algoritmom DFS in globino, ki jo iterativno podaljšujemo. ID kombinira prednost iskanja v globino in iskanja v širino. ID in BFS sta funkcionalno enaka. ID vedno najprej najde najkrajšo pot. Prostorsko je ID manj zahteven od BFS, saj pri ID ni potrebno voditi seznama vseh generiranih sosedov. Časovna zahtevnost ID je $O(b^d)$, prostorska zahtevnost $O(bd)$, kjer je b faktor vejanja grafa, d pa globina najbližjega končnega stanja.

3.3 Hevristično preiskovanje

Če je prostor stanj velik, govorimo o problemu kombinatorične eksplozije. Vseh stanj ni mogoče preiskati v realnem času, zato se moramo zadovoljiti s preiskovanjem majhne podmnožice celotnega prostora stanj. Za preiskovanje uporabljamo hevristične ocene, s katerimi omejujemo in usmerjamo iskanje v smeri najbolj obetavnih vozlišč. Za hevristično cenilko uporabljamo funkcijo $f(n)$, ki ocenjuje težavnost vozlišča n. Funkcija $g(n)$ je ocena cene najboljše poti od začetnega vozlišča do vozlišča n. Funkcija $h(n)$ je ocena cene optimalne poti od vozlišča n do prvega končnega vozlišča. Za lažjo predstavo si poglejmo sliko 4.



Slika 4: Ilustracija funkcij $h(n)$ in $g(n)$.

Pri hevristični oceni govorimo tudi o dopustnosti. Algoritem je dopusten (angl. admissable), če vedno najde optimalno rešitev. Vsaka hevristična funkcija $h(n)$, ki ne precenjuje razdalje do cilja je dopustna. Če predpostavimo, da je za vsako vozlišče n v prostoru stanj $h^*(n)$ cena optimalne poti od n do najbližjega končnega vozlišča, je algoritem dopusten, če velja:

$$\forall n : h(n) \leq h^*(n)$$

Med hevurističnimi preiskovalnimi algoritmi lahko omenimo požrešno usmerjeno iskanje (angl. greedy best-first search) ter algoritem A*. Požrešno usmerjeno iskanje uporablja hevristično cenilko $f(n) = h(n)$, medtem ko A* uporablja funkcijo $f(n) = g(n) + h(n)$.

Trivialna, a neuporabna hevristična funkcija je $h(n) = 0$, ki spremeni algoritem A* v preiskovanje v širino.

3.3.1 Požrešno preiskovanje

Požrešno preiskovanje z dobro hevristično funkcijo pohitri preiskovanje, vendar ne najde vedno optimalne rešitve. V preiskovanju se poglablja v najboljše hevristično ocenjeno vozlišče, sicer pa deluje po principu DFS.

3.3.2 A*

Delovanje algoritma A* je podobno požrešnemu preiskovanju, le da pri hevristični oceni upošteva tudi že prehojeno pot, kar pomeni, da vedno najde optimalno rešitev (če ta obstaja), če je hevristična ocena $h(n)$ dopustna.

Težava algoritma A* je prevelika poraba pomnilnika, saj se pri preiskovanju vsa generirana vozlišča ohranjajo, da lahko izberemo najbolj perspektivno. Izboljšave algoritma A* porabijo manj pomnilnika, a še vedno zagotavljajo optimalnost rešitve. Med izboljšave spada algoritem iterativnega poglabljanja A* (angl. iterative-deepening A*, IDA*), ki namesto povečevanja globine iskanja povečuje vrednost hevristične ocene $h(n)$. Nasledniki vozlišča se generirajo, če imajo hevristično oceno manjšo od določene meje. Ko imajo vsa generirana vozlišča višjo oceno od določene meje, se meja za razvoj poddrevesa popravi in preiskovanje se začne ponovno iz korena. Drug algoritem, ki temelji na A* je Recursive best-first search (RBFS). RBFS shrani vrednost vseh naslednikov na trenutni poti. Preiskovanje nadaljuje v najboljše ocenjenem generiranem vozlišču, za mejo generiranja naslednikov vzame naslednjega najboljšega soseda. Če so hevristične ocene naslednikov nad to mejo, se v prednika zabeleži vrednost najboljšega naslednika (najbolj perspektiven list iz poddrevesa), generirani nasledniki pa se zavržejo. Meja za razvoj poddrevesa se po potrebi popravi na drugega najboljšega soseda in generirajo se nasledniki najboljše ocenjenega lista v drevesu.

3.4 Minimaks

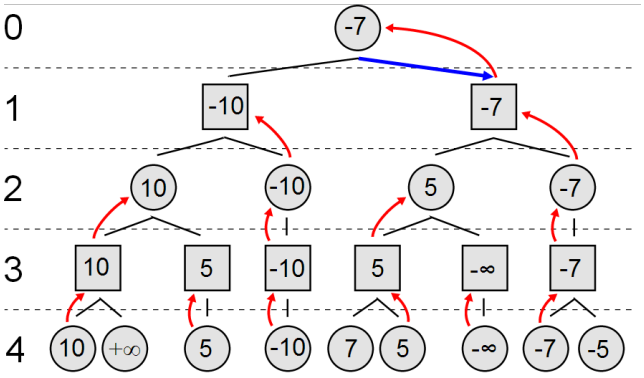
Ko poznamo osnove preiskovanja dreves, lahko razširimo preiskovanje na igre dveh ali več igralcev.

Minimaksov teorem pravi, da za vsako igro z dvema igralcema, ničelno vsoto in končno mnogo strategijami obstaja vrednost V in mešana strategija za vsakega igralca, tako da je

glede na strategijo drugega igralca najboljši možen izkupiček za prvega igralca V in glede na strategijo prvega igralca, najboljši možen izkupiček za drugega igralca $-V$ [12].

Minimaks preiskuje tako, da maksimizira igralčeve možnosti za zmago (dobiček) in minimizira nasprotnikove možnosti za zmago. Algoritem je rekurziven, v vsakem koraku algoritma sta izmenično na potezi igralec in nasprotnik. Vrednost stanja se določi šele v listu drevesa (eksaktno, če je vozlišče končno, ali hevristično, če je preiskovanje omejeno v globino), ta vrednost se prenaša nazaj po preiskovalnem drevesu. Pri vračanju vrednosti se upošteva maksimum naslednikov, če je na potezi igralec, ali minimum naslednikov, če je na potezi nasprotnik.

Primer delovanja algoritma minimaks lahko vidimo na sliki 5. Vozlišča okrogle oblike predstavljajo poteze igralca, ki želi maksimizirati svoj dobiček. Vozlišča kvadratne oblike predstavljajo poteze nasprotnika. Vrednost v vozliščih predstavljajo vrnjene vrednosti. V tem primeru imata oba igralca na voljo dve možni potezi, globina preiskovanja pa je omejena na 4.



Slika 5: Primer preiskovanja z algoritmom minimaks [13].

Algoritem ovrednoti vsak list v drevesu (četrti nivo) s hevristično funkcijo, ocene vozlišč so prikazane na sliki 5. Poteze, kjer je igralec zmagal, so označene s pozitivno neskončnostjo $(+\infty)$, poteze, kjer je zmagal nasprotnik pa z negativno neskončnostjo $(-\infty)$. Na tretjem nivoju se vrednosti listov vračajo v prednike vozlišč. Ker je bil na tretjem nivoju na potezi nasprotnik, se upošteva minimum naslednikov, torej za najbolj levo poddrevo na tretjem nivoju se upošteva $\min(10, \infty)$, torej se v vozlišče shrani vrednost 10. Na drugem nivoju je bil na potezi maksimirajoči igralec in tam se upošteva maksimum generiranih naslednikov. Za najbolj levo poddrevo se upošteva $\max(10, 5)$ in v vozlišče se shrani vrednost 10. To se ponavlja, dokler se algoritem ne vrne v koren drevesa. Vrednost v korenu predstavlja potezo, ki bi jo igralec moral izbrati, da **minimizira** svojo **maksimalno** možno izgubo. V tem primeru je ta poteza označena z modro puščico.

Algoritem minimaks ima časovno kompleksnost $O(b^d)$. Algoritem je opisan s spodnjo psevdokodo [14]:

```
function minimax(vozlišče, globina, max)
    if globina = 0 or vozlišče = končno
        return hevristična vrednost (vozlišče)
    if max
        najboljša vrednost := -∞
        for each naslednik vozlišča
            val := minimax(naslednik, globina - 1, FALSE)
            najboljša vrednost := max(najboljša vrednost, val)
        return najboljša vrednost
    else
        najboljša vrednost:= +∞
        for each naslednik vozlišča
            val := minimax(naslednik, globina - 1, TRUE)
            najboljša vrednost := min(najboljša vrednost, val)
        return najboljša vrednost

(* Začetni klic za igralca *)
minimax(začetno stanje, maksimalna globina, TRUE)
```

3.4.1 Negamaks

Poenostavitev algoritma minimaks je negamaks [15] [16], variacija algoritma, ki se zanaša na lastnost ničelne vsote (opisana v poglavju Osnove preiskovanja) pri igrah dveh igralcev. Algoritem predvideva, da je vrednost nekega stanja pri igralcu enaka negativni vrednosti nasprotnika, kar lahko opišemo s spodnjo enačbo.

$$\max(a, b) = -\min(-a, -b)$$

Ta lastnost poenostavi implementacijo algoritma, ker ni potrebno ločeno obravnavati potez igralca in nasprotnika, saj je vrednost naslednikov, če je na potezi nasprotnik, le negirana vrednost vozlišča. Psevdokoda algoritma je prikazana spodaj.

```
function negamax(vozlišče, globina, predznak)
    if globina = 0 or končno vozlišče
        return predznak * hevristična vrednost (vozlišče)
    najboljša vrednost := -∞
    foreach naslednik vozlišča
        val := -negamax(naslednik, globina - 1, -predznak)
        najboljša vrednost := max(najboljša vrednost, val)
    return bestValue

vrednost := negamax(začetno stanje, maksimalna globina, 1) (* Začetni klic *)
```

3.4.2 Alfa-beta rezanje

Minimaks in negamaks sta izčrpni preiskovalni metodi, ki preiščeta celoten preiskovalni prostor (do določenega nivoja), čeprav so lahko nekatere poteze tako slabe, da se jih ne splača preiskovati. Algoritem alfa-beta rezanja (α-β) rešuje ta problem, saj preiskovanje v algoritmu minimaks ustavi, ko je trenutno vozlišče slabše od že preiskanega. S tem se znatno poveča hitrost algoritma in v enakem času je možno doseči večjo globino, oziroma pogled v naprej (angl. lookahead). Zaradi izločanja poddreves se časovna kompleksnost efektivno zmanjša na $O\left(b^{d/2}\right) = O(\sqrt{b^d})$ [17].

Implementacija algoritmov minimaks ali negamaks se spremeni tako, da se v rekurzivni funkciji prenašata vrednosti α (alfa) in β (beta), ki označujeta najboljše in najslabše preiskano vozlišče. Če je vrednost β manjša od maksimuma vozlišč, ko je na potezi igralec, se preiskovanje naslednikov na tem mestu prekine. Ravno tako se preiskovanje prekine, če je na potezi nasprotnik in je vrednost α manjša od minimuma preiskanih vozlišč. Opis delovanja algoritma je prikazan v spodnji psevdokodi [14].

```
function alphabeta(vozlišče, globina, α, β, max)
     if globina = 0 or končno vozlišče
         return hevristična vrednost (vozlišče)
     if max
         for each naslednik vozlišča
             α := max(α, alphabeta(naslednik, globina - 1, α, β, FALSE))
             if β ≤ α
                 break (* rezanje β *)
         return α
     else
         for each naslednik vozlišča
             β := min(β, alphabeta(naslednik, globina - 1, α, β, TRUE))
             if β ≤ α
                 break (* rezanje α *)
         return β

(* Začetni klic *)
alphabeta(začetno stanje, maksimalna globina, -∞, +∞, TRUE)
```

3.4.3 Preiskovanje glavne inačice

Preiskovanje glavne inačice (angl. principal variation search ali PVS), imenovano tudi NegaScout je dodatna izboljšava algoritma alfa-beta [18]. Temelji na tehniki glavne inačice (angl. principal variation) in se zanaša na to, da so poteze urejene. Glavna inačica je specifično zaporedje potez, ki je najboljše za trenutnega igralca. V praksi se poteze sortirajo glede na prejšnja plitkejša preiskovanja. PVS lahko izvede več rezanj poddreves, če je prvo

preiskano vozlišče dovolj dobro. PVS predpostavi, da je prvo vozlišče glavna inačica, nato pa preveri resničnost predpostavke s preiskovanjem preostalih vozlišč z ničelnim oknom (pri preiskovanju nastavi enako vrednost za α in β). Če je vrednost preiskovanja z ničelnim oknom večja od α in manjša od β, pomeni, da predpostavka ni resnična in prvo vozlišče ni del glavne inačice. V tem primeru mora algoritem ponovno preiskati vsa vozlišča.

PVS ne preišče vozlišč, ki jih lahko izloči alfa-beta, vendar bo pri naključni ureditvi potez PVS porabil več časa kot običajni alfa-beta. Čeprav algoritem ne bo preiskal vozlišč, ki jih tudi alfa-beta ne bi, bi moral veliko že preiskanih vozlišč ponovno preiskati in ovrednotiti, ker pri neuspešnem testu preiskovanja z ničelnim oknom ponovno preišče ista vozlišča.

3.4.4 Ekspektiminimaks

Razširitev algoritma minimaks pri nepopolni informaciji je ekspektiminimaks, ki poleg deterministčnih vozlišč upošteva tudi ostala vozlišča z določeno verjetnostjo. Ta vozlišča vsebujejo pričakovano vrednost naključnega dogodka. Deterministična vračajo min in maks vrednosti, vozlišča z naključnimi dogodki pa povprečimo z njihovo verjetnostjo. Pri maks vozliščih je na potezi igralec, pri min vozliščih nasprotnik, pri vozliščih z verjetnostjo pa se upošteva naključni dogodek (potezo) v igri. Za primer si poglejmo spodnjo psevdokodo, ki upošteva poteze igralca in nasprotnika ter naključne poteze, kot je na primer met kocke [19].

```
function expectiminimax(vozlišče, globina)
    if globina = 0 or končno vozlišče
        return hevristična vrednost vozlišča
    if igralec na potezi
        α := +∞
        foreach naslednik vozlišča
            α := min(α, expectiminimax(naslednik, globina-1))
    else if nasprotnik na potezi
        α := -∞
        foreach naslednik vozlišča
            α := max(α, expectiminimax(naslednik, globina -1))
    else if naključni dogodek
        // Vrni uteženo povprečje vseh naslednikov vozlišča
        // Funkcija za verjetnost P()
        α := 0
        foreach naslednik vozlišča
            α := α + (P(naslednik) * expectiminimax(naslednik, globina -1))
    return α
```

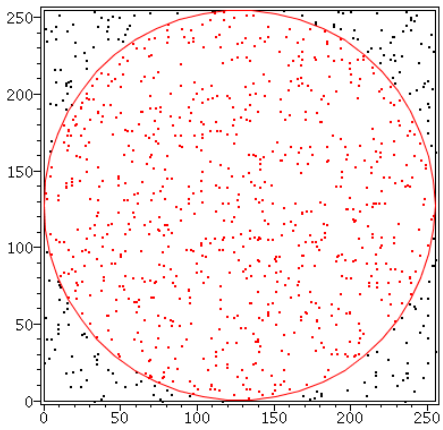
3.5 Metode Monte Carlo

Metode Monte Carlo izvirajo iz statistike in spadajo v razred računalniških algoritmov, ki rezultat ocenijo z večkratnim naključnim vzorčenjem parametričnega prostora. Tehniko uporabljamo v primerih, ko je problemski prostor prevelik, oziroma ko je rezultat z deterministčnim pristopom prezahtevno izračunati. Metode Monte Carlo sledijo vzorcu štirih korakov:

1. Definira se domena možnih vhodnih vrednosti.
2. Naključno se generirajo vrednosti z neko določeno verjetnostno distribucijo.
3. Nad generiranimi vrednostmi se izvrši deterministčen izračun.
4. Rezultati se interpretirajo na podlagi deterministčnih izračunov.

Učinkovitost metode Monte Carlo lahko ilustriramo na primeru računanja vrednosti števila π [10]. V prvem koraku domeno definiramo s kvadratom, ki ima včrtan krog. Stranica kvadrata je enaka premeru kroga, možne vhodne vrednosti pa so vse točke znotraj kvadrata. V drugem koraku se naključno generira veliko število točk v kvadratu. Več kot je generiranih točk, bolj natančen bo končni rezultat. V tretjem koraku se prešteje število točk, ki se nahajajo znotraj včrtanega kroga. V četrtem koraku lahko izračunamo vrednost števila π iz razmerja števila točk znotraj kroga in skupnega števila generiranih točk, ki je približek razmerja med površino kroga (S_c) in površino kvadrata (S_{sq}). Razmerje bi bilo enako, če bi generirali neskončno število točk. Površina kroga je $S_c = \pi(\frac{D}{2})^2$, površina kvadrata $S_{sq} = D^2$, iz česar sledi, da je $\pi = 4\frac{S_c}{S_{sq}}$, torej štirikratnik razmerja površine kroga in kvadrata.

Na sliki 6 prikazujemo delovanje metode Monte Carlo s 1.000 generiranimi točkami in približkom števila $\pi \approx 3.1960$.



Slika 6: Izračun števila π z metodo Monte Carlo in 1.000 točkami.

3.5.1 Preiskovanje dreves Monte Carlo

Preiskovanje dreves Monte Carlo (MCTS) izhaja iz preprostejših in starejših metod Monte Carlo vzorčenja in je nadgradnja teh postopkov. Glavni koncept ostaja enak – program izvaja veliko število naključnih simulacij in na koncu izbere tisto potezo, ki ima najvišji delež zmag. Cilj MCTS je, da izračun konvergira k dobri rešitvi hitreje kot običajni Monte Carlo. To doseže z usmerjanjem simulacij tako, da v drevo dodaja nova, obetavna vozlišča, ki bi bila dosežena prva in večkrat obiskana kot slabša vozlišča [3].

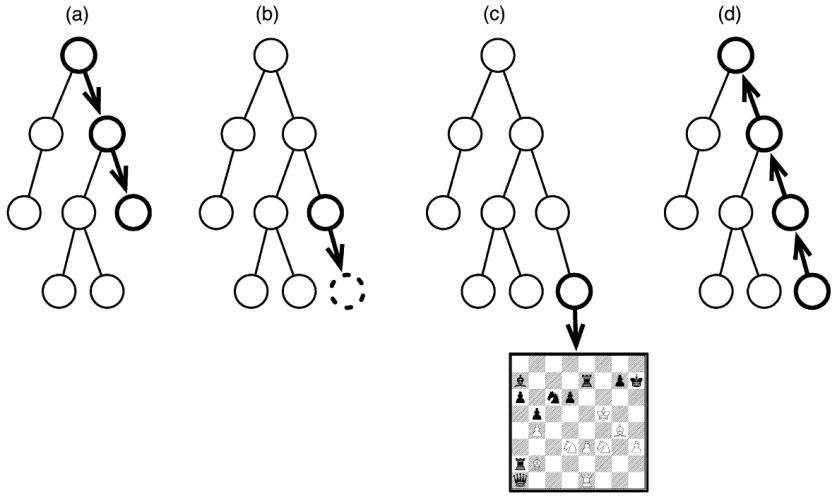
MCTS je iterativna metoda, ki ponavlja štiri korake, dokler je na voljo še kaj časa, oziroma N-krat. Koraki so prikazani na sliki 7.

a) **Izbira** (angl. selection): algoritem izbere vozlišče v listu drevesa, glede na povprečno vrednost vozlišča in število obiskov.

b) **Razširitev** (angl. expansion): algoritem po potrebi doda nova vozlišča v drevo.

c) **Simulacija** (angl. simulation): algoritem simulira preostanek igre iz izbranega vozlišča in vrne vrednost končnega stanja (rezultat igre), ali povprečne vrednosti končnih stanj, če se je simulacija iz vozlišča izvršila večkrat.

d) **Vračanje vrednosti** (angl. backpropagation): vrednost končnega vozlišča se vrača po drevesu navzgor do korena, vsa obiskana vozlišča pa dobijo novo povprečno vrednost.



Slika 7: Prikaz delovanja MCTS [3].

Prednost algoritma MCTS je, da za evalvacijo med preiskovanjem ne potrebujemo hevristične funkcije, saj se simulacija vedno izvrši do konca igre. Ko se omenjeni štirje koraki ponavljajo, se iz korena drevesa vsakokrat izbere naslednika z najvišjo oceno in agent igra s to potezo. MCTS ocenjuje poteze in ne strategije. Implementacija vsakega izmed korakov je poljubna, saj MCTS ne narekuje specifičnih algoritmov za posamezne korake. MCTS ne specificira natančno, katero vozlišče naj se izbere v prvem koraku, na kakšen način se drevo poveča, kako se vršijo simulacije ali kako se vrednosti vračajo nazaj po drevesu. Izbira vozlišč je lahko tudi naključna, vendar dobri rezultati niso zagotovljeni. V [3] navajajo, da pri izbiri navadno povprečenje ne prinaša dobrih rezultatov. Za dobro razmerje med preiskovanjem že preiskanih in novih vozlišč (angl. exploration or exploitation) je priporočljiv algoritem UCT (angl. Upper Confidence bound applied to Trees).

3.5.2 Algoritem UCT

Pri iskanju rešitve z MCTS želimo najti dobro razmerje med preiskovanjem novih vozlišč (eksploracijo) in preiskovanjem obetavnih vozlišč (eksploitacijo). Algoritem, s katerim lahko to razmerje nastavimo, je UCT, ki je definiran z izračunom:

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

V izrazu nastopajo spremenljivke v_i, ki označuje vrednost vozlišča i, koeficient C, ki predstavlja eksperimentalno določeno konstanto razmerja med eksploitacijo in eksploracijo,

množica vozlišč I, število obiskov izhodiščnega vozlišča n_p ter število obiskov trenutno izbranega vozlišča n_i.

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

Koeficient C je priporočeno nastavljen na $\sqrt{2}$, v praksi pa je empirično določen. Za ravnovesje med eksploracijo in eksploitacijo se parameter po potrebi spreminja, tipično na intervalu $\pm\frac{\sqrt{2}}{2}$. Manjša kot bo vrednost konstante C, več časa bo namenjenega preiskovanju obetavnih vozlišč (eksploitaciji), večja kot bo vrednost C, večjo utež bo imela pri preiskovanju eksploracija.

Za primer izračunamo UCT vrednost dveh naslednikov korena drevesa. Prvo vozlišče ima 10 obiskov in vrednost 2, drugo vozlišče pa 1 obisk in vrednost 1. Vrednost C je $\sqrt{2}$, število obiskov korena pa je enak seštevku vseh podrejenih vozlišč. UCT vrednost prvega vozlišča je $2 + \sqrt{2\frac{\ln 15}{10}} \approx 2{,}52$, vrednost drugega vozlišča pa $1 + \sqrt{2 \ln 15} \approx 2{,}64$. V tem primeru bo za razširitev ali simulacijo izbrano drugo vozlišče, ker ima višjo UCT oceno.

Čeprav ima drugo vozlišče pred preiskovanjem nižjo vrednost je favorizirano zaradi manjšega števila obiskov. Do problema, da vozlišče ne bi bilo nikoli preiskano, ne pride, saj bi v izrazu v drugem argumentu enačbe prišlo do deljenja z ničlo (v imenovalcu kvadratnega korena $n_i = 0$), kar je po definiciji neskončno. Še ne preiskana vozlišča imajo avtomatično najvišjo vrednost, zato jih algoritem UCT vsaj enkrat preišče, razen v primeru, ko je $C = 0$. Takrat se upošteva samo prejšnja vrednost vozlišča v_i in je izbira vozlišča za prvo ali drugo fazo MCTS trivialna a neučinkovita, kar smo tudi opazili v naših testih.

Poglavje 4 **Opis rešitve**

Za izdelavo uspešnega agenta smo preizkušali različne pristope in metode opisane v prejšnjih poglavjih. V tem poglavju opišemo glavne pristope in okolje, v katerem smo uspešnost agenta ovrednotili.

4.1 Ekspektiminimaks

Pri izdelavi prototipa igralnega agenta smo na začetku razvoja izhajali iz odprtokodnega projekta Tutegame [11], ki je implementiran z algoritmom ekspektiminimaks. Program deluje tako, da iz vseh neznanih kart (kart v kupčku) generira vse možne konfiguracije kart, ki bi jih lahko igralec imel v roki, nato pa za te konfiguracije izvede izčrpno preiskovanje ekspektiminimaks. Verjetnost negotovih vozlišč se izračuna glede na število kart v kupčku. Pri igri tute ima vsak igralec v roki tri karte, tako da je pred rekurzivnim klicem funkcije ekspektiminimaks trojna for zanka, ki generira vse možne konfiguracije. Časovna kompleksnost pomožne funkcije je $O(n^3)$. Če postopek apliciramo na tršet je časovna kompleksnost pomožne funkcije $O(n^{10})$, kar pomeni, da je za dejanski postopek ekspektiminimaks, ki je časovne kompleksnosti $O(b^d)$, namenjenega veliko manj časa, če želimo igrati v realnem času. Prej v preiskovanju bi se morala klicati hevristična evalvacijska funkcija in prej bi se moralo poglabljanje ustaviti, da bi bila igra izvedljiva v realnem času. Rešitev z ekspektiminimaksom v tršetu ni prinesla dobrih rezultatov, saj je računanje trajalo predolgo. Možna je optimizacija, da bi se pomožna funkcija, ki generira začetne konfiguracije omejila na določeno število konfiguracij. Ker je takšno omejevanje preiskovalnega prostora podobno kot pri metodi Monte Carlo, smo nadgrajevanje rešitve z ekspektiminimaksom opustili in se osredotočili na preiskovanje dreves Monte Carlo.

4.2 MCTS

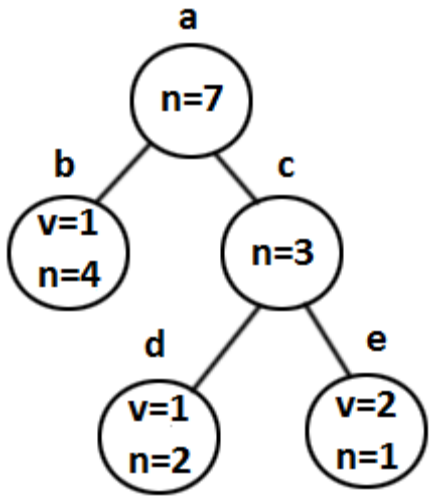
MCTS temelji na večkratnih simulacijah, ki hevristične evalvacijske funkcije ne potrebuje, saj se simulacija vedno izvrši do konca igre. V naši rešitvi smo za iterativni klic MCTS definirali metodo start(), kjer generiramo naslednike iz začetnega stanja. To stanje je lahko začetek igre ali kasnejše, vedno pa predpostavimo, da je v korenu preiskovalnega drevesa na potezi igralni

agent. Nasledniki so veljavne poteze, ki jih agent lahko naredi, in ta korak je razširitev preiskovalnega drevesa. Iz generiranih naslednikov nato izberemo najboljšega v metodi select() z algoritmom UCT. Za vsakega naslednika izračunamo UCT vrednost po naslednji formuli:

$$uctValue = \frac{node.value}{node.numVisits + \varepsilon} + C\sqrt{\frac{\log(allVisits + 1)}{node.numVisits + \varepsilon}} + rand(0,1) * \varepsilon$$

Konstanta ε je majhna vrednost (10^{-6}) in jo prištevamo k imenovalcu ulomka, da ne bi prišlo do deljenja z ničlo. Prvi argument enačbe je povprečje prej izračunanih UCT vrednosti – seštevek vseh UCT vrednosti trenutnega vozlišča pri vsaki iteraciji deljen s številom obiskov. V drugem argumentu nastopa koeficient eksploracije, s konstanto C, ki smo jo pri testiranju spreminjali. Z zadnjim argumentom formule ($rand(0,1) + \varepsilon$) naključno izberemo naslednika v primeru enakosti.

Za primer na sliki 8 izračunamo UCT vrednost vseh treh listov v drevesu (vozlišča *b*, *d* in *e*). Število obiskov je označeno z n, vrednost vozlišč pa z v. Ker se pri MCTS računa UCT samo za liste drevesa, smo na našem primeru vrednosti vmesnega vozlišča *c* in korena *a* izpustili, število obiskov teh vozlišč pa je enako seštevku obiskov vseh podrejenih vozlišč.



$$UCT(b) = \frac{1}{4} + \sqrt{2\frac{\log(7+1)}{4}} \cong 1{,}27$$

$$UCT(d) = \frac{1}{2} + \sqrt{2\frac{\log(7+1)}{2}} \cong 1{,}94$$

$$UCT(e) = 2 + \sqrt{2\log(7+1)} \cong 4{,}04$$

Slika 8: Primer izračuna UCT.

Za najboljše ocenjenega naslednika (iz primera na sliki 8 je to vozlišče *e* z oceno 4,04) se izvrši simulacija – naključno odigrana igra do konca. V vsaki iteraciji, dokler je informacija nepopolna, nasprotniku določimo naključne karte iz kupčka, tako da ima nasprotnik v roki 10 kart. Med igro dobivamo informacijo o jemanju kart iz kupčka, kar lahko izkoristimo. Karti,

ki jo ima nasprotnik zagotovo v roki, lahko določimo večjo težo, kot kartam, ki jih ne poznamo zagotovo, in to upoštevamo v fazi simulacije. Ko je simulacija končana, rezultat shranimo v vsa obiskana vozlišča v drevesu z izrazom:

$$v = w * \delta + (1 - \delta) * s$$

Spremenljivka w je binarna in ima vrednost 0 ali 1. Označuje, ali je igra prinesla zmago (1) ali ne (0). Spremenljivka s označuje število točk v igri. Konstanta δ je empirično določena, glede na teste, ki smo jih opravili in predstavlja utež, koliko naj se v algoritmu UCT upošteva binarna zmaga in koliko število točk.

Ker nasprotniku zaradi nepopolne informacije določimo naključne karte, vsa vozlišča po koncu simulacije pod prvim nivojem zavržemo. Prvi nivo predstavljajo poteze agenta, ocena uspešnosti pa postopoma konvergira k neki vrednosti. Vsako iteracijo simulacijo izvršimo iz vozlišča na prvem nivoju, določenega z izrazom UCT, dokler je na voljo še kaj časa. Ko se preiskovanje zaključi, za potezo izberemo vozlišče iz prvega nivoja z najboljšo oceno.

Za dobre rezultate moramo imeti čim več MCTS iteracij. Rešitev bo hitreje konvergirala z dobro izbiro parametrov ter z optimalno simulacijo. Boljše kot igralca igrata v fazi simulacije, bolj natančna bo ocena izbrane poteze. Idealno bi v fazi simulacije morali imeti perfektno igro, kar bi zahtevalo izčrpno preiskovanje. To vzame veliko časa, za čim večje število MCTS iteracij pa želimo, da se simulacija izvede čim hitreje. Naiven način je, da v fazi simulacije mečemo naključne veljavne karte. Za izboljšavo smo vpeljali hevristiko z različnimi obtežitvami kart (angl. weighted random), tako da imajo močnejše karte manjšo težo.

Preiskovanje smo časovno omejili na 2 sekundi, v tem času se izvede nad 10.000 MCTS iteracij. Dve sekundi z dodatnim časovnim zamikom za prenos podatkov iz platforme za testiranje sta dovolj za občutek, da je igra odzivna, kot bi potekala proti človeškemu igralcu.

4.3 Minimaks

V 10 rundi igre, ko imamo že na voljo popolno informacijo, začnemo z izčrpnim preiskovanjem. Implementirali smo algoritem minimaks z alfa-beta rezanjem. Časovna omejitev je enaka kot pri preiskovanju MCTS, na dve sekundi. Pri globini 9 algoritem porabi manj časa, za globino 10 pa uporabimo enostavno hevristično evalvacijsko funkcijo – število točk pridobljenih v igri. Igralca v igri nista nujno izmenično na potezi, prvi igralec v novi rundi je tisti, ki je prejšnjo rundo zmagal. To pri implementaciji minimaksa upoštevamo in pri rekurzivnem klicu za zadnji argument, ne glede na to, ali računamo min() ali max(), določimo ali je naslednji na potezi igralec ali nasprotnik.

4.4 Platforma za testiranje

Testiranje vpliva parametrov metode MCTS smo izvajali v programskem jeziku java. Razvili smo razred MCTSTest, kjer smo izvajali veliko število simulacij z različnimi parametri, simulirali igro proti različnim pristopom – MCTS proti naključnemu igralcu, MCTS proti minimaks ter MCTS proti MCTS z drugačnimi parametri. V simulacijah so se vrednosti samodejno spreminjale in rezultati povprečili po večurnem izvajanju.

Testiranje proti človeškim igralcem smo izvajali na spletnem portalu Briskula.si. Na portalu je registriranih 27.500 uporabnikov, skupno je bilo od ustanovitve portala odigranih že več kot 10 milijonov iger [20]. Spletna aplikacija je razdeljena na strežniški in uporabniški del. Strežniški del poganja TCP (angl. Transmission Control Protocol) vtičnik implementiran v programskem jeziku PHP z vso programsko logiko. Uporabniški del v Flashu se preko vtičnika XMLSocket poveže na strežnik in z njim komunicira. Tako lahko igralci igrajo eden proti drugemu v živo. Igralni agent je samostojna PHP aplikacija, ki se preko vtičnika poveže na strežnik. Ta aplikacija je ogrodje za komunikacijo s strežnikom, inteligentne agente zaradi hitrosti poganjamo v javi. Ko je igralni agent na potezi, iz ukazne vrstice požene javanski program, ki izvaja preiskovanje. Parametre igre pošlje preko argumentov ukazne vrstice v formatu JSON (JavaScript Object Notation). JSON zapis vsebuje podatke o tem, katere karte ima agent v roki, znane karte nasprotnika, preostale karte v igri, število točk, karte na mizi in podatek, ali je igra tršet ali kifameno. Preostale karte v igri in znane karte nasprotnika so lahko enake. Primer JSON zapisa parametrov igre je prikazan v spodnjem izpisu.

```
{
     "hand": ["K 3", "K KAVAL", "D 2", "D AS", "B 7"],
     "hiscards": ["D KAVAL", "D KRALJ", "D 3", "K 2", "K 7"],
     "unknown": ["D KAVAL", "D KRALJ", "D 3", "K 2", "K 7"],
     "inplay": [],
     "points": 18,
     "cfm": false
 }
```

Iz prejetih podatkov program v javi zgradi stanje in začne s preiskovanjem. Če je razlika množice preostalih kart in znanih kart prazna, požene minimaks, drugače MCTS. Program po koncu izvajanja rezultat – karto z najboljšo oceno – vrne na standardni izhod (angl. standard output), kjer ga PHP aplikacija prebere in posreduje na strežnik.

4.5 Zasnova podatkovne baze

Za ocenjevanje uspešnosti agenta imamo definirano podatkovno bazo odigranih iger, potez in človeških igralcev. Igralci so rangirani po uspešnosti, to je po deležu zmag in številu odigranih iger. Podatki se shranjujejo v bazi MySQL.

Shema podatkovne baze je prikazana na sliki 9. Tabela uporabnikov ima sledeče atribute: uporabniška šifra (angl. User ID), uporabniško ime, skupno število odigranih iger, skupno število zmag in skupno število točk. Izid posamezne igre shranimo za vsakega igralca posebej. V igri dveh igralcev na koncu igre vstavimo dva nova vnosa v bazo. Shema odigranih iger ima atribute: časovni žig, uporabniška šifra, binarni atribut ali je uporabnik zmagal ali ne, število točk in vrsta igre (tršet ali kifameno). Vsako potezo (odvrženo karto) shranjujemo v bazo za kasnejše podatkovno rudarjenje in iskanje dobrih strategij. Vsaka odvržena karta predstavlja en vnos v bazi z atributi: uporabniška šifra, šifra igre, množica igralčevih kart v roki, množica znanih nasprotnikovih kart, množica kart na mizi, množica spravljenih kart za vsakega igralca posebej, odvržena karta (poteza), prva karta na mizi, število točk, ki jih ima igralec, časovni žig, število igralcev in vrsta igre (tršet ali kifameno).

Slika 9: Shema podatkovne baze.

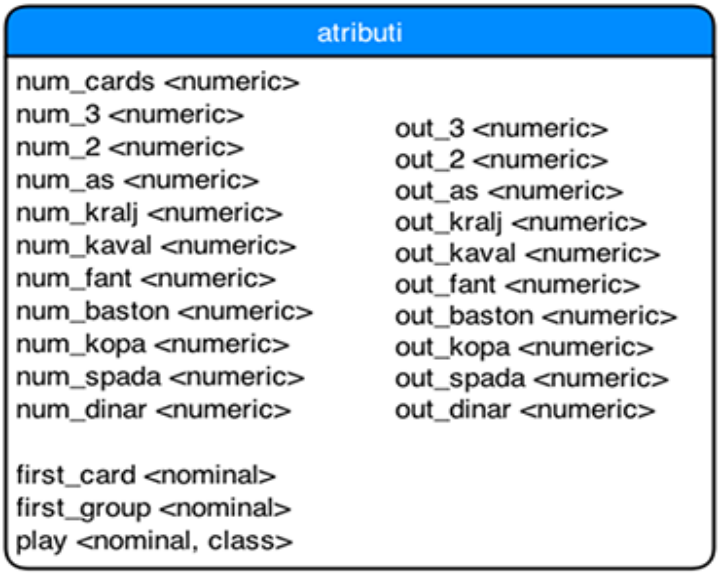
Z zbiranjem podatkov o potezah smo začeli 21. oktobra 2012. Zbrane podatke smo uporabili za izboljšavo preiskovanja MCTS. Od 21. 10. 2012 se je akumuliralo več kot 16 milijonov zabeleženih potez. Iz te baze smo izbrali igre tršet dveh igralcev, nato pa še vse poteze dobrih igralcev, ki imajo več kot 10.000 odigranih iger in nad 50% deležem zmag.

```
SELECT uid FROM briskula WHERE trs_game > 10000 AND trs_win/trs_game > 0.5
```

Tem pogojem ustreza 1800 registriranih uporabnikov in s filtriranjem potez teh igralcev nam je ostalo več kot 1 milijon shranjenih potez. Izbrane podatke smo razdelili na dva dela, na del, kjer je bil igralec prvi na potezi, ter na del, kjer je bil igralec drugi na potezi. Preostalih 500.000 vrstic v posamezni tabeli smo naključno razdelili na učno in testno množico.

4.6 Izraba znanja človeških igralcev

Iz podatkov smo izdelali višjenivojske atribute. Iz baze MySQL smo izvozili podatke in jih s PHP skripto pretvorili v novo shemo, prikazano na sliki 10. Z izbranimi atributi na abstrakten način opišemo stanje igre. Znanje človeških igralcev izrabimo tako, da med igro glede na abstraktne atibute poiščemo podobna stanja iz baze iger, ki so jih odigrali dobri igralci. Višjenivojski atributi so številčni, a diskretni. Za atribute smo določili število kart v roki, število trojk, dvojk, asov, kraljev, kavalov in fantov ter število kart iz vsake skupine. Atribute, ki predstavljajo enake podatke, smo določili tudi za karte izven igre. Pri bazi potez, kjer je bil igralec drugi na potezi, smo dodali še dva atributa – skupino prve padle karte in številko. Razred za klasifikacijo je vržena karta (niz).



Slika 10: Shema pretvorjene podatkovne baze.

Iz baze podatkov smo med preiskovanjem želeli poiskati podobne situacije, ki so jih odigrali dobri igralci, ter zabeležiti njihovo akcijo. Pri preiskovanju med gradnjo drevesa smo podobne akcije verjetnostno obtežili. Algoritem, ki rešuje ta problem, je k-najbližjih sosedov (angl. k-Nearest Neighbors, k-NN). Algoritem k-NN je ena enostavnejših metod strojnega

učenja in spada med metode lenega učenja (angl. instance-based learning, lazy learning). To pomeni, da za klasifikacijo uporablja primere (instance) iz učne množice, ki so shranjeni v spominu [21]. Kompleksnejše metode iz učnih podatkov zgradijo generaliziran model [22].

Pri delu smo uporabili odprtokodno knjižnico za podatkovno rudarjenje v javi – Weka [23]. Na voljo je javanski JAR paket (angl. java archive), ki ima programski vmesnik za klic funkcij v javi (API – Application Programming Interface). Z Weka API lahko enostavno naložimo podatke in jih klasificiramo.

Weka ne podpira razredov z več kot enim atributom (angl. multi-label learning), zato je naš razred specifična karta. Podatke za učenje smo iz datoteke CSV (angl. Comma Separated Value) pretvorili v format ARFF (angl. Attribute-Relation File Format), ki ga Weka podpira. Pretvorjeno datoteko preberemo z Weko in jo uporabimo za klasifikator po načelu najbližjih sosedov. Ko želimo poiskati najbližje sosede, naredimo novo instanco, določimo atribute in poženemo iskanje. Primer branja podatkov, gradnje klasifikatorja in iskanje najbližjih sosedov je prikazan v spodnji kodi.

```
// Preberemo podatke
BufferedReader reader = new BufferedReader(new FileReader("dataset.arff"));
Instances dataset = new Instances(reader);

// Naučimo klasifikator
LinearNNSearch knn = new LinearNNSearch(dataset);

// Za hitrejše iskanje lahko uporabimo k-d drevo
KDTree knn = new KDTree();
knn.setInstances(dataset)

// Naredimo instanco, za katero želimo poiskati sosede
Instance instance = new Instance(dataset.numAttributes());

// Instanci določimo atribute
instance.setValue(dataset.attribute("n"), context.getSize());

// Najbližje sosede poiščemo s kNN
Instances nearestInstances = knn.kNearestNeighbours(instance, 3);
```

Iskanje najbližjih sosedov ima časovno kompleksnost $O(dN)$ [24], kjer je N število primerov v učni množici in d število atributov. Iskanje najbližjih sosedov se izvede večkrat v vsaki iteraciji MCTS, zato smo morali iskanje optimizirati, da se izvede v krajšem času. Za pohitritev iskanja najbližjih sosedov smo uporabili k-d drevo. K-d drevo je drevesna podatkovna struktura, ki shranjuje točke iz k-dimenzionalnega prostora. Listi drevesa vsebujejo primere iz učne množice, vozlišča pa delijo preiskovalni prostor na dva dela. Časovna kompleksnost poizvedbe za najbližjega soseda v k-d drevesu je $O(d \log N)$ [24].

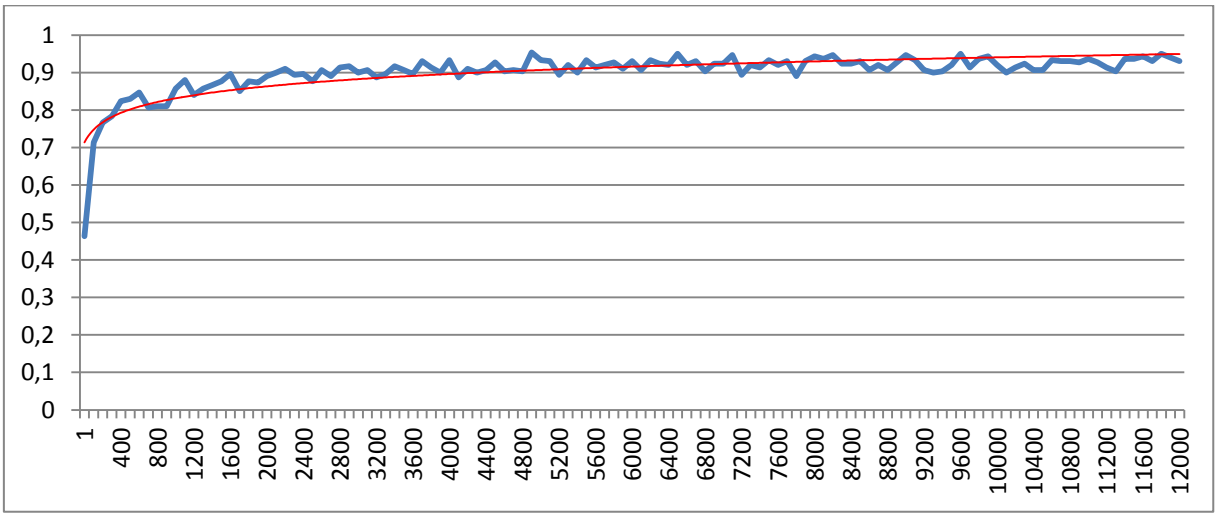
Iskanje najbližjih sosedov smo uporabili v fazi simulacije, ko se preiskovalno drevo zgradi do končnega stanja. Pri gradnji drevesa se po definiciji MCTS generirajo naključni nasledniki. Glede na najdene sosede smo podobnim naslednikom določili večjo verjetnost. Iz razreda najbližjega soseda smo poiskali najbolj podobno karto v naboru veljavnih kart. Privzeta utež vseh kart je 1. Če smo našli karto, ki se popolnoma ujema z razredom (enaka barva in številka), smo ji utež povišali za 5. Kartam, ki so v enaki skupini kot karta iz razreda najbližjega soseda, smo povišali utež za 2. Kartam z enako številko, kot karta iz razreda, smo utež povišali za 1.

Poglavje 5 **Evalvacija**

Uspešnost agenta smo merili z igranjem proti različnim pristopom in proti človeškim igralcem. V tem poglavju opišemo rezultate posameznih testiranj.

5.1 Testiranje agenta proti naključnemu igralcu

Testiranje smo najprej opravili z igranjem MCTS agenta proti naključnemu igralcu (ko je na potezi, vrže igralec naključno veljavno karto). Začeli smo z vplivom spreminjanja števila MCTS iteracij na uspešnost, kot je prikazano v sliki 11. Na vodoravni osi prikazujemo število iteracij, na navpični osi pa povprečen delež zmag. Pri vsakem testiranju smo število iteracij povečali za 400, agent pa je odigral 300 iger. Vsaka točka na grafu predstavlja eno testiranje. Z modro črto je predstavljena uspešnost pri vsaki meritvi, z rdečo črto pa logaritemska trendna črta iz pridobljenih podatkov.

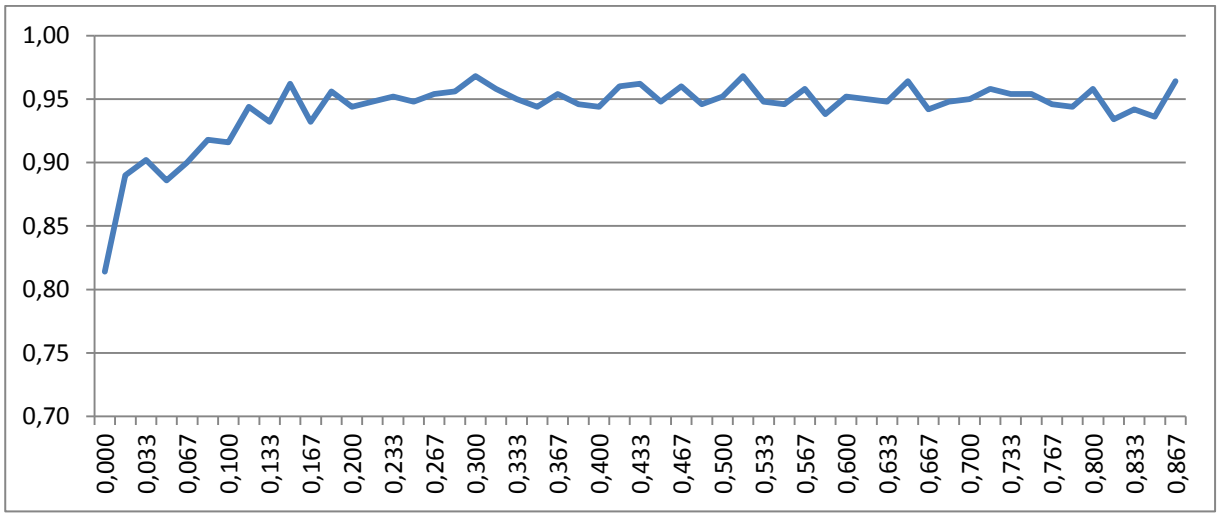


Slika 11: Uspešnost agenta glede na število MCTS iteracij v igri proti naključnemu igralcu.

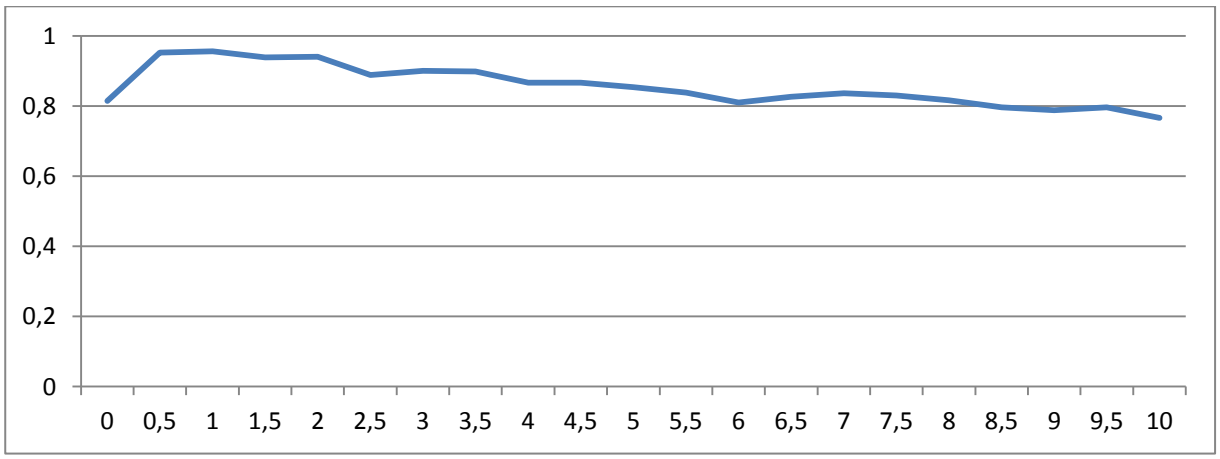
Z nizkim številom iteracij je uspešnost enaka kot pri naključnemu agentu (50% delež zmag), že pri 1.000 iteracijah pa uspešnost naraste nad 80%. Vidimo lahko, da pri velikem številu iteracij uspešnost ne konvergira k 100% deležu zmag, ampak je zaradi variance, ki je lahko posledica slabih kart, ustaljeno na približno 95% uspešnost pri 10.000 iteracijah. Povprečna

uspešnost vseh meritev je 90,14%, z minimumom 46,3%, maksimumom 95,3% in standardnim odklonom 5,56%.

Parameter razmerja med eksploracijo in eksploitacijo C smo najprej testirali na intervalu $C = [0, 1]$, pri čemer smo za vsako meritev odigrali 500 iger s 15.000 MCTS iteracijami (slika 12). Ugotovili smo, da je C dovolj robusten in da nima velikega vpliva na rezultate. Pri $C = 0$ (samo eksploitacija) je bil delež zmag 81,4% in se je nato ustalil na 95% pri $C \geq 0{,}2$ z maksimalno uspešnostjo 96,8%. Povprečna uspešnost pri testiranju je bila 94,24% s standardnim odklonom 2,54%. Testiranje smo ponovno pognali na intervalu $C = [0, 10]$, izvedli 21 meritev s korakom 0,5 (slika 13). Rezultati so podobni, parameter C se je izkazal za zelo robustnega, uspešnost iz 95,6% na 90% pade šele pri $C = 2{,}5$ in nato postopoma pada do 76,6% pri $C = 10$. Povprečna uspešnost vseh meritev je 85,5%, standardni odklon pa 5,73%.



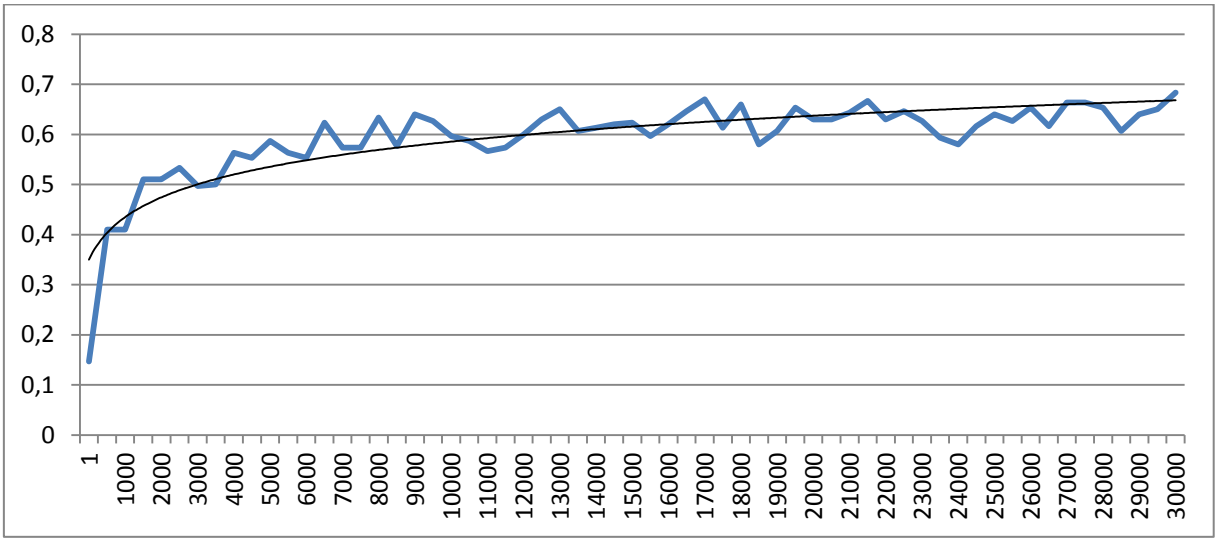
Slika 12: Uspešnost glede na vrednost parametra C na intervalu [0, 1].



Slika 13: Uspešnost glede na vrednost parametra C na intervalu [0, 10].

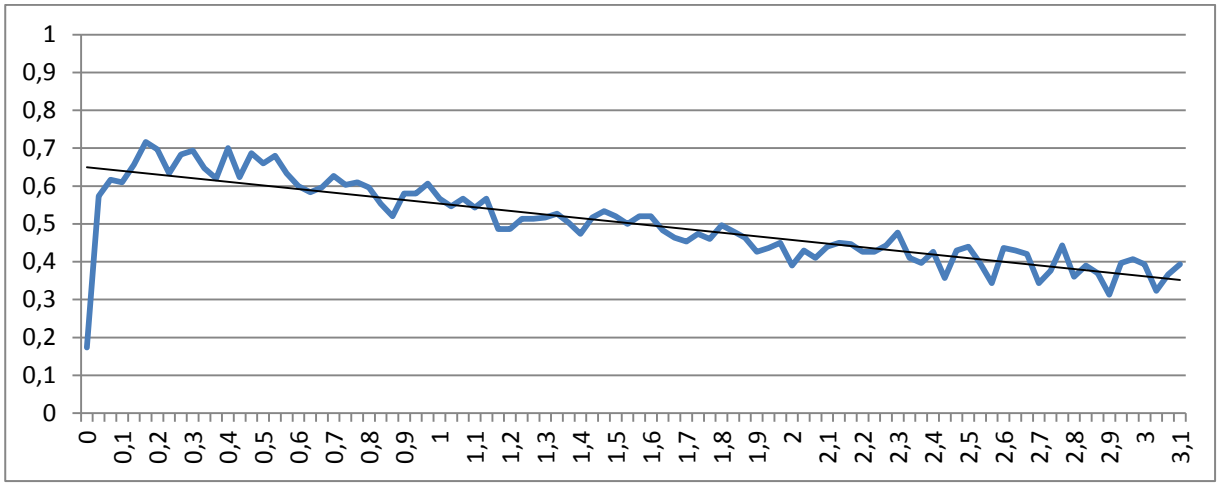
5.2 Testiranje agenta proti samemu sebi

Jasnejšo sliko vpliva parametrov na uspešnost dobimo, ko agent igra proti samemu sebi. V testiranju smo merili uspešnost glede na število MCTS iteracij. En agent je imel pri vsaki meritvi 2.000 iteracij, drugi pa je število postopoma povečeval na intervalu $[1, 30000]$ s korakom 1.000. Za vsako meritev smo odigrali 300 iger. S samo eno MCTS iteracijo je imel agent proti drugemu agentu uspešnost 14,67%. Uspešnost naraste na 50% pri 2.000 iteracijah in konvergira proti 70%. Maksimalna uspešnost pri meritvah je bila 68,3%, povprečje 59,43% s standardnim odklonom 8,07%. Iz grafa na sliki 14 lahko vidimo, da zelo veliko število iteracij (nad 10.000) nima velikega vpliva na uspešnost. Število iteracij med 5.000 in 10.000 je dovolj za uspešnega agenta. Grafu je dodana logaritemska trendna črta, ki konvergira k 0,7.



Slika 14: Uspešnost MCTS agenta, ki je igral proti samemu sebi z različnim številom iteracij.

Povsem drugačne rezultate opazimo na sliki 15, kjer smo vizualizirali vpliv parametra C na uspešnost, ko MCTS agent igra proti samemu sebi. Grafu je dodana linearna trendna črta. Prvi agent je imel C empirično določen na $\sqrt{2}$, drugega smo testirali na intervalu $[0, 3]$ s korakom 0,03. V tem testiranju je bila uspešnost največja, ko je agent igral z večjo mero eksploitacije in manj eksploracije (C med 0,3 in $\sqrt{2} \cong 1{,}41$). Uspešnost pada linearno s C in je pri večji meri eksploracije vedno slabša. Opravili smo 94 meritev, minimalna uspešnost je bila 17,3% pri $C = 0$, maksimalna 71,67% pri $C = 0{,}17$ in povprečna uspešnost 50% s standardnim odklonom $\sigma = 10{,}6\%$.



Slika 15: Uspešnost MCTS agenta, ki je igral proti samemu sebi z različnimi vrednostmi parametra C.

Rezultate smo poskusili izboljšati z različnim načinom ocenjevanja vozlišč v zadnji fazi MCTS po formuli, opisani v poglavju 4.2. V tej formuli ($v = w * \delta + (1 - \delta) * s$) smo spreminjali razmerje med binarnim rezultatom zmage in številom točk za določitev vrednosti vozlišča. Spreminjali smo vrednost spremenljivke δ na intervalu $[0, 1]$ s korakom 0,1. Manjša vrednost δ je pomenila, da so bila vozlišča ocenjena s številom točk pridobljenih v igri. Večja vrednost δ je pomenila, da je bilo vozlišče točkovano binarno, ali je poteza prinesla zmago ali ne. Izvedli smo dva ločena testa, pri čemer je agent MCTS igral proti samemu sebi, enkrat je imel nasprotnik fiksno določeno $\delta = 0$ (tabela 1), drugič pa $\delta = 1$ (tabela 2). Za vsako meritev je bilo odigranih 300 iger pri 4.000 MCTS iteracijah. Velikih razlik v rezultatih ne opažamo, je pa tendenca k boljši uspešnosti pri večji vrednosti δ, kar pomeni večjo težo za binarno ocenjevanje vozlišč. Ko je nasprotnik igral z $\delta = 1$ je bila uspešnost uravnotežena pri 50% šele, ko je tudi agent imel $\delta = 1$.

Po koncu metode MCTS lahko iz preiskanega drevesa najboljšo potezo izberemo na različne načine. V vseh omenjenih testih smo končno potezo izbrali z algoritmom UCT, kot da ponovno izvajamo prvo fazo MCTS iz korena – izbiro (angl. selection). Potezo smo v enem testu poskusili izbrati tudi tako, da smo iz korena drevesa izbrali naslednika z najvišjim seštevkom vrednosti izidov igre (v), v drugem testu pa naslednika z najvišjo povprečno vrednostjo izidov igre $\left(\frac{v}{numVisits}\right)$. V teh testih je nasprotni agent potezo izbral z UCT. V prvem testu je bila uspešnost agenta 70% pri 2.000 MCTS iteracijah, $\delta = 1$ in $C = \sqrt{2}$ in vzorcem 100 iger. Uspešnost z višanjem iteracij konvergira proti 50%. Pri 10.000 iteracijah je bila uspešnost 64 z enakimi parametri. Pri drugem testu je bila uspešnost agenta slabša, 7%.

δ	Povprečna uspešnost
0	0,55
0,1	0,56
0,2	0,63
0,3	0,59
0,4	0,63
0,5	0,65
0,6	0,64
0,7	0,64
0,8	0,68
0,9	0,68
1	0,71

Tabela 1: Povprečna uspešnost proti nasprotniku z $\delta = 0$. $min = 0{,}55; max = 0{,}71; \sigma = 0{,}05$

δ	Povprečna uspešnost
0	0,28
0,1	0,33
0,2	0,39
0,3	0,38
0,4	0,42
0,5	0,50
0,6	0,45
0,7	0,46
0,8	0,44
0,9	0,46
1	0,55

Tabela 2: Povprečna uspešnost proti nasprotniku z $\delta = 1$. $min = 0{,}28; max = 0{,}55;\ \sigma = 0{,}07$

Pri naslednjem testu smo pri naključni igri (odvržejo se naključne karte) karte obtežili z njihovimi točkami. Izvedli smo 10 meritev, pri vsakem testu je bilo odigranih 1.000 iger s 5.000 MCTS iteracijami, utež pa smo spreminjali na intervalu $[0, 1]$ s korakom 0,1. Opazili smo, da takšna obtežitev kart ne vpliva na uspešnost MCTS agenta. Povprečna uspešnost je bila 0,51, minimum 0,48, maksimum 0,54 in standardni odklon 0,02. Podobni so tudi rezultati tega testa, kjer je bil nasprotnik naključni igralec. Povprečna uspešnost 89% z minimumom 85,9%, maksimumom 90,6% in standardnim odklonom 1,46%.

Za referenco merila uspešnosti in vpliva variance na igro smo izvedli še test dveh MCTS agentov z enakimi parametri, kjer je eden igral tršet, drugi pa kifameno. Ko en igralec igra na zmago, drugi na poraz, smo pričakovali rezultat 100% uspešnosti, a je povprečje 500 odigranih iger s 3.000 MCTS iteracijami agenta za tršet pokazalo 98,4% uspešnost.

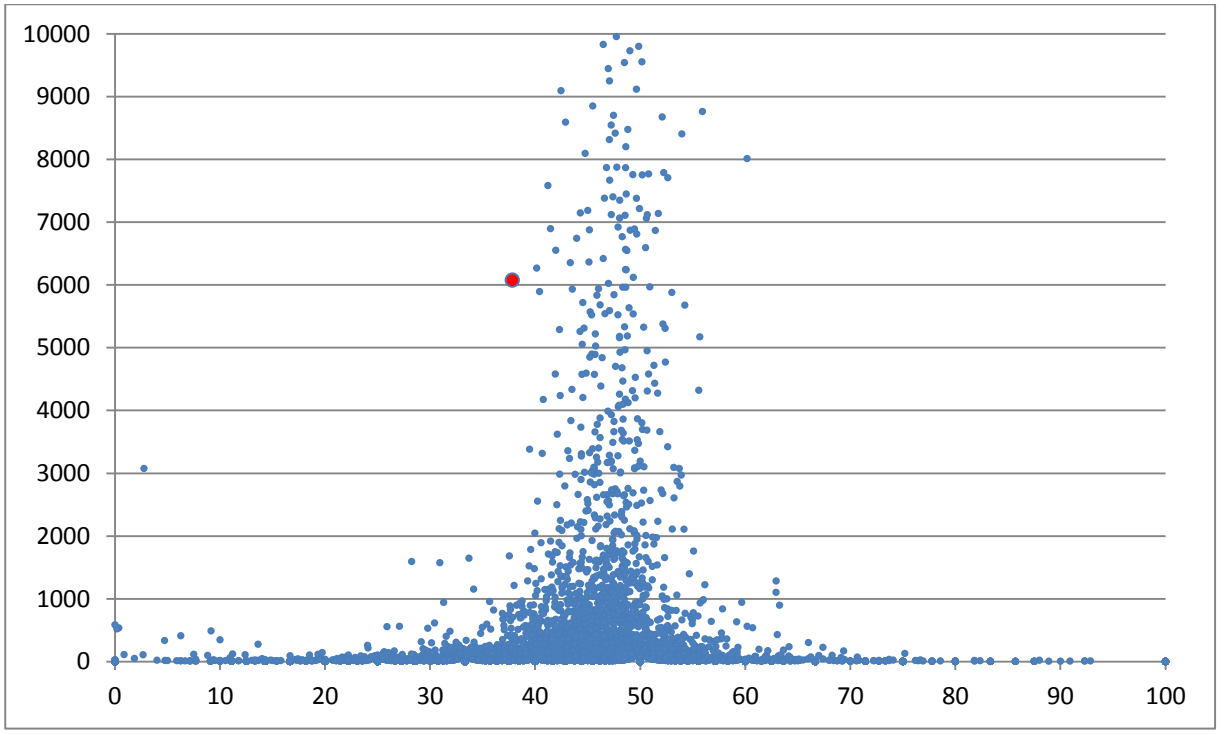
5.3 Testiranje MCTS agenta proti minimaksu

Od trenutka v igri, ko imamo popolno informacijo, smo izvedli test uspešnosti MCTS agenta proti enakemu MCTS agentu, ki v zadnjih desetih potezah preiskuje z algoritmom minimaks. Iz 50% smo uspešnost izboljšali na 64% - agent z minimaksom je zmagal 646 iger od 1.000 odigranih.

5.4 Testiranje agenta proti človeškim igralcem

Testiranje smo izvedli na spletnemu portalu Briskula.si, kjer je agent igral proti človeškim igralcem. Od 6045 iger jih je zmagal 2280, kar pomeni, da je njegova uspešnost 37,7%. V

povprečju je v igri dobil 15,7 točk od 18 potrebnih za zmago ($\sigma = 4{,}99$). Za primerjavo – povprečen delež zmag vseh registriranih igralcev, ki so odigrali vsaj eno igro na portalu je 38,9%. Uspešnost je višja pri igralcih, ki so odigrali več iger na portalu. Povprečna uspešnost igralcev, ki so odigrali vsaj 100 iger je 45,3%, pri 1.000 igrah 47,4%, pri 10.000 igrah 49,4% in postopoma narašča. Igralec je na portalu odigral v povprečju 212 iger tršeta, vendar so posamezni igralci odigrali že več kot 100.000 iger. Slika 16 na grafu prikazuje povprečno uspešnost igralcev. Na vodoravni osi je uspešnost izražena v odstotku deleža zmag. Na navpični osi je število iger. Povprečna uspešnost iz grafa je 40,3% z $\sigma = 18{,}7\%$. Iz grafa smo odstranili ekstreme – igralce, z več kot 10.000 odigranimi igrami. Manjkajočih 146 igralcev iz grafa ima v povprečju uspešnost 49,4% z $\sigma = 3{,}28\%$.



Slika 16: Povprečna uspešnost igralcev na Briskula.si, ki so odigrali med 1.000 in 10.000 iger.

Povprečna uspešnost igralcev ni natančno 50% ker spletni portal upošteva tudi predane igre – ko igralec zapusti igro pred koncem. V tem primeru se mu v statistiko zabeleži ena izgubljena igra.

Naš agent se v teh podatkih umešča v spodnji del seznama uspešnih agentov in je na grafu prikazan z rdečo piko. Med 640 igralci, ki so odigrali vsaj 1.000 iger tršeta je na lestvici uspešnosti agent na 637 mestu.

5.5 Testiranje agenta s pristopom k-NN

Časovna kompleksnost izčrpne metode preiskovanja najbližjih sosedov je povzročila, da smo morali parametre MCTS metode znatno zmanjšati, da smo lahko izvedli testiranje. Za določitev uspešnosti v vzorcu 100 odigranih iger smo preiskovanje MCTS zmanjšali na 100 iteracij. Učno množico smo zmanjšali na približno 100.000 primerov. S temi parametri je testiranje potekalo 12 ur. Iskanje najbližjih sosedov v k-d drevesu je hitrejše, zato smo testiranje ponovno izvedli na vzorcu 1.000 iger s 300 iteracijami MCTS. Pri vsakem testiranju je en agent v med simulacijo generiral naslednike z enako verjetnostjo, drugi pa obteženo glede na najbližje sosede.

Na obeh vzorcih smo imeli enak rezultat, 50% uspešnost. Preiskovanje s pristopom k-NN z izbranimi parametri do zdaj tako ni prineslo izboljšav.

Poglavje 6 **Zaključek**

Rezultat magistrskega dela je prototip inteligentnega agenta za igranje igre s kartami tršet. V zaključku povzamemo narejeno, izpeljemo glavne sklepe, opravimo kritično analizo dela ter opišemo ideje za izboljšave.

Agenta smo implementirali v javi in ga uporabili na spletnem portalu Briskula.si, ki omogoča igranje kart v živo proti drugim igralcem. Prototip agenta smo najprej izdelali s pristopom ekspektiminimaks, ki ni dosegal dobrih rezultatov. Možne izboljšave za ekspektiminimaks so temeljile na vzorčenju, zato smo prešli na metodo Monte Carlo, ki je takoj pokazala uspešne rezultate. Uspešnost smo izboljševali z empiričnim nastavljanjem parametrov metode preiskovanja dreves Monte Carlo (MCTS) in na koncu še dodatno izboljšali agenta s preiskovanjem minimaks, ko v igri nastopi popolna informacija.

Igralni agent na spletnem portalu Briskula.si dosega rezultate pod povprečjem, vendar je uspešnost zadovoljiva, saj so z njim kot s soigralcem zadovoljni tudi nadpovprečno dobri igralci.

> *[Agenti] so dobri, zaključek je odličen. Prav zanimivo je opazovati, kako se borijo za zadnjo [potezo]*.
>
> – igralka s 132.000 odigranimi igrami in 52,3% deležem zmag.

Povprečna uspešnost agenta na portalu je 37,7%, povprečna uspešnost vseh igralcev pa 38,9%. Iz referenčnega testa uspešnosti smo ugotovili, da je vpliv variance v igri dovolj velik, da 100% uspešnost praktično ni mogoča.

Skozi magistrsko nalogo smo preučili preiskovanje pri igrah z nepopolno informacijo. MCTS je uspešna metoda delnega preiskovanja velikega determinističnega problemskega prostora, dobre rezultate pa dosega tudi pri preiskovanju z nepopolno informacijo. Robustnost metode ponuja veliko možnosti za izboljšave in specifične prilagoditve problemu.

Cilj magistrske naloge je bil izdelati konkurenčnega igralca tršeta in uporaba opisanih pristopov je pripomogla k dosegu cilja. Rezultati magistrske naloge nam kažejo, da so opisane metode uporabne in uspešne pri podobnih problemih.

Rezultat magistrske naloge je uspešen robotski igralec igre tršet. Izdelana rešitev pa še vedno ponuja možnosti za izboljšave. Agent z minimaks pristopom se dobro obnese v delu igre s popolno informacijo, ker je preiskovanje izčrpno in je igra, če ne zmanjka časa, popolna. Popolna informacija v igri nastopi prepozno, da bi bil slab začetek igre še rešljiv. Iz naših rezultatov opažamo, da je najpomembnejši del igre na začetku, takrat pa je količina informacij na voljo najmanjša. Agenta bi lahko izboljšali z obširnejšim preiskovanjem in boljšim izkoriščanjem hevristik naučenih iz podatkovne baze. Pričakovali smo izboljšanje rezultatov s pristopom k-NN, vendar jih nismo opazili. Razlogi za to so lahko optimizacijski. Iskanje k-NN je računsko zahteven problem in za izvedbo testiranja v razumnem času smo morali znatno zmanjšati učno množico. Optimizacija iskanja s k-d drevesi je bila hitrejša, a ne dovolj, da bi jo lahko uporabili pri testiranju proti človeškim igralcem.

MCTS je možno izboljšati s paralelizacijo – z vzporednim izvajanjem algoritma na več nitih ali procesih. Na ta način se lahko izkorišča večjedrne ali večprocesorske računalnike in se v enakem času lahko preišče mnogo večji del prostora. Paralelizacija je možna na tri načine:

1. Paralelizacija listov – vzporedno se lahko izvaja več simulacij iz istega lista drevesa, če je implementacija metode takšna, da se v eni iteraciji izvede več simulacij.

2. Paralelizacija korena – gradijo se lahko neodvisna drevesa iz korena in poteza se izbere glede na vsa generirana drevesa.

3. Paralelizacija drevesa – vzporedno se lahko gradi isto preiskovalno drevo, pri čemer je potrebno paziti na sinhronizacijo, na primer z zaklepanjem.

S paralelizacijo bi na začetku igre lahko zgradili več dreves, pri katerih ima nasprotnik različne karte v roki. Na ta način bi lahko drevo razširjali, ker bi preiskovanje posameznega drevesa potekalo tako, kot da imamo popolno informacijo.

Dodatne izboljšave so možne pri izrabi znanja pridobljenega iz človeških iger. Potrebno bi bilo odkriti boljše in enostavnejše atribute, ki vsebujejo čim večji del informacij. Abstraktno bi morali predstaviti tudi razred, ker klasifikacija za specifično karto iz nabora atributov ni možna, saj se navezuje na množico kart, ki jih ima igralec v roki in na pravila igre. Atribute bi morali oceniti glede pomembnosti ter uporabiti iskanje v uteženem prostoru.

Iskanje najbližjih sosedov je možno pohitriti in izboljšati s približnimi metodami. Hitrost iskanja najbližjih sosedov s k-d drevesi pri visoko-dimenzionalnih podatkih ni boljša od navadnega linearnega preiskovanja [24]. Med metode približnega iskanja najbližjih sosedov spada krajevno zgoščevanje (angl. locality-sensitive hashing, LSH).

Poglavje 7 **Literatura**

[1] P. I. Cowling, C. D. Ward in E. J. Powley, „Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering," *IEEE Transactions On Computational Intellingence and AI in Games,* zv. 10, oktober 2012.

[2] D. Whitehouse in E. J. Powley, „Determinization and Information Set Monte Carlo Tree Search for," *IEEE Conference on Computational Intelligence and Games (CIG'11),* pp. 87-94, 2011.

[3] P. Ciancarini in G. P. Favini, „Monte Carlo tree search in Kriegspiel," *Artificial Intelligence,* zv. 174, p. 70–684, april 2009.

[4] J. Long, N. R. Sturtevant, M. Buro in T. Furtak, „Understanding the Success of Perfect Information," *Proceedings of the Twenty-Fourth AAAI Conference on Artificial Intelligence (AAAI-10),* pp. 134-140, 2010.

[5] C. Browne, „Monte Carlo Tree Search," Dostopno na http://mcts.ai/. Dostop: 25 avgust 2014.

[6] P. Grošelj, „Obravnavanje medsebojnega vpliva biološke raznovrstnosti in ekonomske aktivnosti s pomočjo teorije iger," 19 februar 2001. Dostopno na http://www.drustvo-informatika.si/fileadmin/dsi2001/sekcija_operacijske_raziskave/groselj.doc. Dostop: 1 september 2014.

[7] „Nash Equilibrium - Investopedia," Dostopno na http://www.investopedia.com/terms/n/nash-equilibrium.asp. Dostop: 1 september 2014.

[8] T. Furtak in M. Buro, „Minimum Proof Graphs and Fastest-Cut-First Search Heuristics,“ *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence,* pp. 492-498, 2009.

[9] M. Lapajne, „*Program za igranje taroka z uporabo drevesnega preiskovanja Monte-Carlo*,“ diplomsko delo. Fakulteta za računalništvo in informatiko, 2011.

[10] A. Babič, „*Drevesno preiskovanje po metodi Monte Carlo pri igri poker*,“ diplomsko delo. Fakulteta za računalništvo in informatiko, 2013.

[11] R. Rosas, „Design of Spanish Cards game known as Tute,“ Google Code, 4 maj 2008. Dostopno na: https://code.google.com/p/tutegame/. Dostop: 15 marec 2013.

[12] M. J. Osborne in A. Rubinstein, „*A Course in Game Theory*,“ MIT, Cambridge, MA, 1994.

[13] N. Nogueira, „File:Minimax.svg,“ 4 december 2006. Dostopno na: http://commons.wikimedia.org/wiki/File:Minimax.svg. Dostop: 1 september 2014.

[14] „Minimax search and Alpha-Beta Pruning,“ Dostopno na: http://www.cs.cornell.edu/courses/cs312/2002sp/lectures/rec21.htm. Dostop: 15 september 2014.

[15] G. T. Heineman, G. Pollice in S. Selkow, „Path Finding in AI,“ *Algorithms in a Nutshell,* p. 213–217, O'Reilly Media, Inc, 2008.

[16] M. Luštrek, „*Patologija v hevrističnih preiskovalnih algoritmih*,“ doktorska disertacija. Fakulteta za računalništvo in informatiko, 2007.

[17] C. Grosan in A. Abraham, „*Intelligent Systems: A Modern Approach*,“ Springer Science & Business Media, 2011.

[18] B. Boskovič in J. Brest, „Chess Program Umko,“ *Elektrotehniški vestnik,* zv. 78(3), pp. 153-158, 2011.

[19] D. Michie, „Game-playing and game-learning automata,“ *Advances in Programming and Non-Numerical Computation,* In L. Fox (urednik): pp. 183-200, 1966.

[20] „Briskula.si,“ 2008. Dostopno na: http://www.briskula.si. Dostop: 1 september 2014.

[21] S. Russell in P. Norvig, „*Artificial Intelligence: A Modern Approach,*“ druga izdaja, p. 733, Pearson Education, 2003.

[22] S. Džeroski in K. Driessens, „Combining Model-Based and Instance-Based Learning for First Order Regression,“ *Proceedings of the 22nd International Conference on Machine Learning,* 2005.

[23] „Weka 3 - Data Mining with Open Source Machine Learning Software in Java,“ Dostopno na: http://www.cs.waikato.ac.nz/. Dostop: 1 september 2014.

[24] M. A. Kibriya in E. Frank, „An Empirical Comparison of Exact Nearest Neighbour Algorithms,“ 2007. Dostopno na: http://www.cs.waikato.ac.nz/~ml/publications/2007/KibriyaAndFrankPKDD07.pdf. Dostop: 15 september 2007.

[25] I. Bratko, „*Prolog in umetna inteligenca*,“ Fakulteta za računalništvo in informatiko, 1997.

[26] B. Merry, J. Gain in P. Marais, „*Accelerating kd-tree searches for all k-nearest neighbours*,“ tehnično poročilo. Department of Computer Science, University of Cape Town, 2013.