Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Andreja Kropivšek

Primerjava algoritmov za računalniško igranje namizne igre Gobblet

diplomsko delo

na univerzitetnem študiju
računalništva in informatike

Mentor: doc. dr. Polona Oblak

Ljubljana, 2014

Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01977 / 2013
Datum: 13.12.2013

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

Kandidat: **ANDREJA KROPIVŠEK**

Naslov: **PRIMERJAVA ALGORITMOV ZA RAČUNALNIŠKO IGRANJE NAMIZNE IGRE GOBBLET**
COMPARISON OF ALGORITHMIC SOLVERS FOR THE BOARD GAME GOBBLET

Vrsta naloge: DIPLOMSKO DELO UNIVERZITETNEGA ŠTUDIJA

Tematika naloge:

Drevesno preiskovanje Monte-Carlo je novejši algoritem, ki daje obetavne rezultate na področju igranja iger in je alternativa klasičnemu algoritmu minimaks. V delu razvijte računalniške različice namizne igre Gobblet. Zapišite dobro ocenjevalno funkcijo algoritma minimaks. Hkrati analizirajte, kako se metoda Monte-Carlo obnaša pri različnih parametrih in rezultate primerjajte z minimaks agentom.

Mentor:
doc. dr. Polona Oblak

Fakulteta za računalništvo in informatiko • UNIVERZA V LJUBLJANI • 2

Dekan:
prof. dr. Nikolaj Zimic

Izjava o avtorstvu diplomskega dela

Spodaj podpisana Andreja Kropivšek, z vpisno številko **63050161**, izjavljam, da sem avtorica diplomskega dela z naslovom:

Primerjava algoritmov za računalniško igranje namizne igre Gobblet

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelala samostojno pod mentorstvom doc. dr. Polone Oblak,
- 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 10. marec 2014 Podpis avtorja:

Zahvaljujem se mentorici doc. dr. Poloni Oblak za vso podporo, strokovno pomoč in dragocene nasvete med pisanjem diplomske naloge. Hvala tudi Rudiju vso podporo ter vsem prijateljem, ki so v času nastajanja nalogo prebrali in komentirali. Zadnja zahvala velja družini, ki mi je med študijem stala ob strani.

Kazalo

Povzetek

Gobblet je zanimiva namizna igra, ki združuje elemente križcev in krožcev z igro spomina. Igra obstaja v dveh različnih velikostih igralnega polja, 3×3 ter 4×4. Cilj naše naloge je bil razviti računalniško različico obeh iger in zanju implementirati uspešne igralce. Za ta namen smo raziskali dva algoritma, ki veljata kot priljubljena izbira za reševanje tovrstnih problemov umetne inteligence: minimaks in drevesno preiskovanje Monte Carlo.

Predstavili smo delovanje algoritma minimaks, njegovo pohitritev z alfa-beta rezanjem in opisali razvito hevristično ocenjevalno funkcijo. Vse to smo uporabili za razvoj računalniškega igralca, tako imenovanega agenta minimaks. Naleteli smo tudi na problem patologije minimaksa, ki smo ga opisali na našem primeru. Kot nasprotnika smo razvili agenta drevesnega preiskovanja Monte Carlo, predstavili njegovo delovanje in natančno opisali metode določanja vrednosti parametrov, pri katerih je bila njegova igra najbolj uspešna.

V rezultatih smo predstavili izide igranja agenta minimaks proti agentu drevesnega preiskovanja Monte Carlo, za obe različici igre. Kot zanimivost smo dodali tudi rezultate uspešnosti igranja proti obstoječi implementaciji igre Gobblet.

Ključne besede: Gobblet, namizna igra, algoritem minimaks, alfa-beta rezanje, hevristična ocenjevalna funkcija, metoda drevesnega preiskovanja Monte Carlo, patologija minimaksa

Abstract

Gobblet is a compelling board game that combines elements of classical tic-tac-toe game with a memory game. It exists in two different sizes of boards: 3×3 and 4×4. The goal of this work was to implement a computer version of the game and to research possible solvers that would implement AI players. We picked two most commonly used solutions for these kinds of problems in artificial intelligence: algorithm minimax and Monte Carlo tree search method, also known as MCTS.

We examined the minimax algorithm and its optimisation with the alpha-beta pruning. For an estimation of a given state we developed our own heuristic evaluation function. We used it to develop our first AI player called the Minimax bot. We also examined the pathology in the minimax algorithm that we faced during our work. As an opponent MCTS bot was presented. All the steps of MCTS were introduced and we explained the method used for tuning the best values for its parameters.

For both board sizes, the outcomes of play-outs between MCTS and the minimax bot are presented. As an interesting comparison we also present results of play-outs of our bots with an existing implementation.

Keywords: Gobblet, board game, minimax, alpha-beta pruning, heuristic evaluation function, Monte Carlo tree search, MCTS, pathology in minimax

Poglavje 1

Uvod

1.1 Motivacija

Klasični izziv umetne inteligence so namizne igre. Pol stoletja raziskovalnih naporov je bilo potrebno, da je računalnik premagal svetovnega prvaka v šahu. Gonilna sila v tem podvigu je bil algoritem minimaks. Kljub zmagi računalnika nad človekom v šahu pa še vedno obstajajo igre, v katerih je človek močnejši nasprotnik. Takšen primer je igra go. Vendar se v zadnjih letih tudi v goju kaže hiter napredek izredno uspešnih agentov, ki uporabljajo drevesno preiskovanje Monte Carlo (ang: Monte Carlo tree search, v nadaljevanju MCTS).

V nalogi smo želeli raziskati, kako se minimaks in MCTS obneseta v isti igri. Za izhodišče smo izbrali dokaj novo namizno igro Gobblet, v kateri igralca tekmujeta, kateri bo na plošči prvi uspel postaviti svoje figure v vrsto. Vsebuje elemente postavljanja, premikanja in prekrivanja figur.

1.2 Cilji

Cilj naloge je razviti računalniško različico namizne igre Gobblet in ji dodati uspešne računalniške igralce t.i. agenta umetne inteligence. Za uresničitev tega cilja bomo uporabili algoritem minimaks in drevesno preiskovanje Monte

Carlo.

1.3 Zgradba dela

- V 1. poglavju predstavimo motivacijo, cilje in zgradbo naloge.
- V 2. poglavju sledi opis igre Gobblet in njena umestitev v teoriji iger.
- 3. poglavje je predstavitev algoritma minimaks, alfa-beta rezanja in hevristične ocenjevalne funkcije.
- 4. poglavje se ukvarja z metodo Monte Carlo, drevesnim preiskovanjem Monte Carlo in postopkom za določanje vrednosti parametrov, ki so uporabljeni pri slednjem.
- 5. poglavje je predstavitev naše implementacije igre Gobblet za operacijski sistem Windows in agente, ki uporabljajo predstavljene algoritme za igranje.
- 6. poglavje je prikaz rezultatov igranja igre Gobblet z našimi agenti.
- V 7. poglavju so podane ugotovitve naloge.

Poglavje 2

Ozadje

2.1 Gobblet

Gobblet je novejša namizna igra za dva igralca, ki jo je zasnoval Thierry Denoual in leta 2001 izdal založnik Blue Orange. Je kombinacija spominskih in abstraktnih strateških iger. Spominja sicer na znano igro križci in krožci, od nje pa se razlikuje po spominskem elementu, saj se figure lahko premikajo in manjše prekrivajo z večjimi. Izvor imena je angleška beseda *goblet*, ki pomeni čaša in spominja na obliko figur.

Igra Gobblet obstaja v dveh različicah (glej sliko 4.1), ki se razlikujeta po velikosti igralne plošče. Prva je izšla igra za ploščo v velikosti 4×4, dve leti kasneje pa še različica 3×3, ki meri na mlajše igralce.

Slika 4.1: Levo je primer namizne igre Gobblet 3×3, desno pa Gobblet 4×4. Vir slike: [3] [2].

2.1.1 Pravila igre Gobblet 3×3

Sestavine Gobbleta 3×3 so igralna plošča, ki ima 3×3 polja, to je 9 polj in 12 dvobarvnih igralnih figur. Vsak igralec igra s svojo barvo. Barve se razlikujejo skladno z različnimi izdajami igre, mi smo izbrali rdečo modro kombinacijo. Figure so treh velikostih, vsako velikost predstavljata 2 figuri.

Zmaga igralec, ki prvi postavi svoje 3 figure v vidno vrsto horizontalno, vertikalno ali diagonalno. Z vsako potezo se sme postaviti na ploščo novo figuro ali premakniti figuro, ki je že na plošči. Uporabljena figura mora ostati na plošči. Po premiku mora biti končna lokacija na plošči različna od začetne. Figura zasede prazno mesto na plošči ali pa prekrije manjšo figuro, ki je že na plošči, in to ne glede na barvo. Tako so lahko na enem polju tri figure različnih velikosti. Dovoljeni so premiki samo zgornje nepokrite figure, prav tako samo zgornje figure štejejo za zmago. V primeru, da premik naše figure odkrije manjšo figuro, ki odpre tri v vrsto za nasprotnika, tedaj zmaga nasprotnik, razen če pravkar dvignjena figura prekrije kakšno drugo iz zmagovalnega zaporedja. Po dotiku figure je treba potezo odigrati izključno z dotaknjeno figuro. Preverjanje, kaj se skriva pod zgornjo figuro, ni dovoljeno. Zmaga igralec, ki prvi oblikuje ravno vidno vrsto iz treh figur svoje barve.

2.1.2 Pravila igre Gobblet 4×4

Igro Gobblet 4×4 sestavlja igralna plošča 4×4. Vsak igralec ima na voljo 12 figur. Figure se razlikujejo po štirih različnih velikostih, vsako velikost predstavljajo 3 figure. Cilj igre je razvrstiti štiri figure naše barve v ravno vrsto. Osnovna pravila premikanja figur so enaka pravilom za verzijo 3×3.

Obstajajo pa dodatne omejitve, ki ne veljajo za premike, ampak le pri prvem postavljanju figure na ploščo:

- Pred začetkom igre mora vsak igralec ob igralni plošči urediti svoje igralne figure v 3 identične stolpiče. V vsakem stolpiču so 4 figure, tako da večje pokrivajo manjše.
- Nekatere izdaje igre 4×4 vključujejo tudi pravilo, da je treba figuro iz zunanjega stolpiča, položiti na prazno polje. Za pokrivanje nasprotnikove figure se jo sme uporabiti le takrat, ko ima nasprotnik že 3 figure iste barve v vrsti.

Zmaga prvi igralec, ki postavi 4 figure svoje barve v vidno vrsto.

2.2 Namizne igre in računalništvo

Arheološki dokazi kažejo, da si je človeštvo že zelo zgodaj krajšalo čas z raznovrstnimi namiznimi igrami. Najstarejšo znano igro senet so odkrili v grobnicah predinastijskega Egipta okoli 3500 pr. n. št. [21].Danes še vedno izjemno popularen go je v kitajskih zapisih prvič omenjen davnega leta 548 pr. n. št.

Zamisel, da bi stroj lahko igral igro, se je porodila še pred razvojem modernih računalnikov. V 19. stoletju si je Charles Babbage zamislil avtomat za igranje križcev in krožcev, čeprav ga ni nikoli dokončal, pa je intenzivno raziskoval zmagovalne strategije.

Posledično tudi ni presenetljivo, da je bilo igranje iger prvo raziskovalno področje umetne inteligence. Šah ima v raziskovanju umetne inteligence

in kognitivne znanosti podobno funkcijo kot vinska mušica (*Drosophila*) v genetiki: idealno okolje, v katerem lahko nadzorovano preizkušamo hipoteze o naravi inteligence in računske sheme za inteligentne sisteme.

2.3 Umestitev igre Gobblet

Matematično obravnavo naše igre lahko opišemo s teorijo iger. Teorija iger preučuje odločitve agentov oziroma igralcev, ki s svojimi akcijami medsebojno vplivajo drug na drugega. V teoriji iger Gobblet uvrstimo med deterministične, zaporedne igre za dva igralca, z ničelno vsoto in popolno informacijo z določenim možnim izidom.

- *Deterministične* igre nimajo elementa naključja. Njihovo nasprotje so stohastične igre z elementom naključja, na primer met kocke.
- *Zaporedne* igre so tiste, kjer igralci zaporedoma in v znanem vrstnem redu izvajajo poteze. Razlikujejo se od iger, pri katerih so poteze izvedene sočasno, na primer pri igri kamen, škarje, papir.
- Igre *ničelne vsote* so igre, kjer boljši rezultat enega igralca takoj pomeni slabši rezultat drugih igralcev. Vsoto nič dobimo, če na koncu vse pridobljene točke igralcev seštejemo z vsemi izgubljenimi točkami igralcev. Pri šahu tako zmagovalec dobi $+1$ točko, poraženec -1 točko, če igralca remizirata, ne dobita točk.
- Pri igrah s *popolno informacijo* je trenutno stanje igre v celoti razkrito za vse igralce. Nasprotne so igre z nepopolno informacijo, tako pri pokru ne poznamo kart drugih igralcev.

Igre za dva igralca, ki imajo vse naštete lastnosti, definiramo kot kombinatorne igre. Takšna je tudi obravnavana igra Gobblet. Kombinatorne igre imajo določeno število položajev, ki jih opisuje množica dovoljenih premikov za vsakega igralca. Vsak premik prvega ali drugega igralca vodi v nov položaj, ki se imenuje opcija. Vsako od teh opcij lahko vzamemo kot novo igro zase,

spet opisano z množico pravil za oba igralca.

To s stališča matematične predstavitve igre pomeni, da so vse množice položajev za prvega in drugega igralca dosegljive iz katerekoli danega položaja. Tako je mogoče igro prikazati kot usmerjeno drevo. Usmerjeno drevo definira koren, ki predstavlja začetno stanje igre. Vozlišča so nova stanja igre po premiku, ki ga predstavlja povezava med korenom in vozliščem. Vsako novo stanje igre je koren novemu poddrevesu, ki ima svoja vozlišča novih stanj, do katerih privedejo z dovoljenimi premiki iz korena.

Tradicionalne kombinatorne igre predpostavljajo, da igralec, ki naredi zadnjo potezo, zmaga, nobena poteza pa ne sme biti ponovljena zapored. Temu ustreza Gobblet 3×3, za Gobblet 4×4 pa se lahko pojavijo cikli, zato je bilo treba dodatno definirati remi igre. Igra je neodločena, če igralec začne ponavljati neko potezo, saj bi katerakoli druga poteza pomenila neposredni poraz.

2.4 Znane implementacije

Znana je implementacija igre Gobblet v programskem jeziku Racket [1]. Kljub temu da za igro še ne obstaja strategija, ki bi vodila v zmago, avtorji trdijo, da njihova implementacija zagotavlja zmago za prvega igralca na plošči 3×3. Njihova implementacija bo služila za primerjavo uspešnosti agentov v nalogi. Cilj pa je bil da zmagamo na plošči 4×4.

Poglavje 3

Minimaks

V tem poglavju predstavimo algoritem minimaks, pohitritev z alfa-beta rezanjem in hevristično ocenjevalno funkcijo.

3.1 Izrek minimaks

Izrek minimaks Johna von Neumana iz leta 1928 [20], pravi:

Za vsako igro dveh igralcev z ničelno vsoto in končno mnogo strategijami, obstaja vrednost V in mešana strategija za vsakega igralca, tako da:

- pri dani strategiji za drugega igralca, je najboljši možni rezultat za prvega igralca V, in
- pri dani strategiji za prvega igralca, je najboljši možni rezultat za drugega igralca $-V$.

Ekvivalentno mu strategija prvega igralca zagotavlja rezultat V ne glede na strategijo drugega igralca. Podobno strategija drugemu igralcu zagotavlja rezultat $-V$. Ime minimaks izvira iz dejstva, da vsak igralec poskuša minimizirati maksimalni rezultat za nasprotnega igralca. Ker gre za igre ničelne vsote, pri tem tudi minimizira svojo maksimalno izgubo, oziroma maksimizira svoj minimalni rezultat.

3.2 Iskalno drevo minimaks

V teoriji iger lahko igre kot Gobblet rešujemo kot problem kontratikdornega iskanja, pri katerem imata dve strani nasprotujoče si cilje. Rešujemo ga s pomočjo izreka minimaks in usmerjenega drevesa.

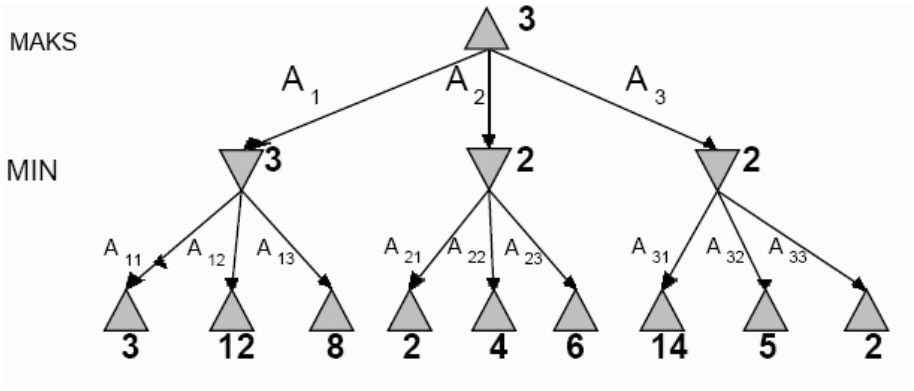
Dva igralca, poimenovana $MAKS$ in MIN, premikata figure, dokler se igra ne konča. Na koncu igre zmagovalec prejme maksimalno število točk, poraženec pa minimalno. Formalno gre za iskalni problem z naslednjimi komponentami: [22]

- *Začetno stanje*je stanje igralne plošče in informacija, kateri igralec je na potezi.
- *Funkcijo novih stanj*, ki vrne množico parov (premik, novo stanje), kjer vsak par predstavlja dovoljen premik in stanje po premiku.
- Test *končnega stanja*, ki preveri, ali je igre konec in sporoči zmagovalca. Stanja v katerih se igra konča, so *končna stanja*.
- *Ocenjevalna funkcija*, ki nam poda numerično vrednost končnega stanja. Pri igrah ničelne vsote je ocena drugega igralca nasprotno enaka oceni prvega.

S preiskovanjem drevesa poskušamo določiti najboljšo potezo. Preiskovanje poteka tako, da dodeljujemo ocene posameznim listom in vozliščem izgrajenega drevesa, pri čemer uporabljamo strategijo *najprej v globino (depth-first)*. Ocenjevalna funkcija poda zgolj oceno vrednosti listov drevesa (končnih stanj). Ocene drugih vozlišč se izračunajo iz ocen neposrednih naslednikov, torej končnih stanj (listov), do katerih vodijo njihove povezave.

Ocene vozlišč se računajo po algoritmu minimaks. Igralca se imenujeta $MAKS$ in MIN. Optimalna rešitev iskalnega problema za igralca $MAKS$ je zaporedje premikov, ki vodi do zanj končnega zmagovalnega vozlišča. Algoritem minimaks predpostavlja, da oba igralca igrata optimalno, če namreč

MIN ne igra optimalno, je $MAKS$ v še boljšem položaju. Primer na sliki 11.1 prikazuje, da so možni premiki za igralca $MAKS$ A_1, A_2, in A_3. Možni odgovori igralca MIN, za potezo A_1, pa so A_{11}, A_{12}, in A_{13} itd. Igra na sliki se konča v dveh potezah.



Slika 11.1: Dva nivoja igralnega drevesa. $\triangle$ predstavljajo vozlišča $MAKS$ (torej da je na potezi $MAKS$ igralec) in vozlišča $\triangledown$ predstavljajo igralca za MIN. Končna vozlišča prikazujejo oceno, ostala vozlišča imajo dodeljene vrednosti po izvajanju algoritma minimaks. Končni rezultat pomeni, da je najboljši premik za $MAKS$ A_1, najboljši odgovor od MIN pa A_{11}. [22]

Postopek ocenjevanja vozlišč je naslednji:

- vozlišče je končno: z ocenjevalno funkcijo izračunamo njegovo vrednost;
- vozlišče je na nivoju MIN: dodelimo mu najmanjšo oceno naslednikov;
- vozlišče je na nivoju MAX: dodelimo mu največjo oceno naslednikov.

Tako se rekurzivno od spodaj navzgor oceni vrednosti vseh vozlišč v poddrevesih, ki izhajajo neposredno iz korena drevesa. Vrednost v korenu predstavlja končno oceno trenutnega položaja. Povezava, ki vodi do vozlišča z isto oceno, pa je izračunana najboljša možna poteza za igralca $MAKS$.

Algoritem minimaks izvaja kompletno preiskovanje drevesa *najprej v globino*. To pomeni, da kadar je maksimalna globina drevesa m in je b

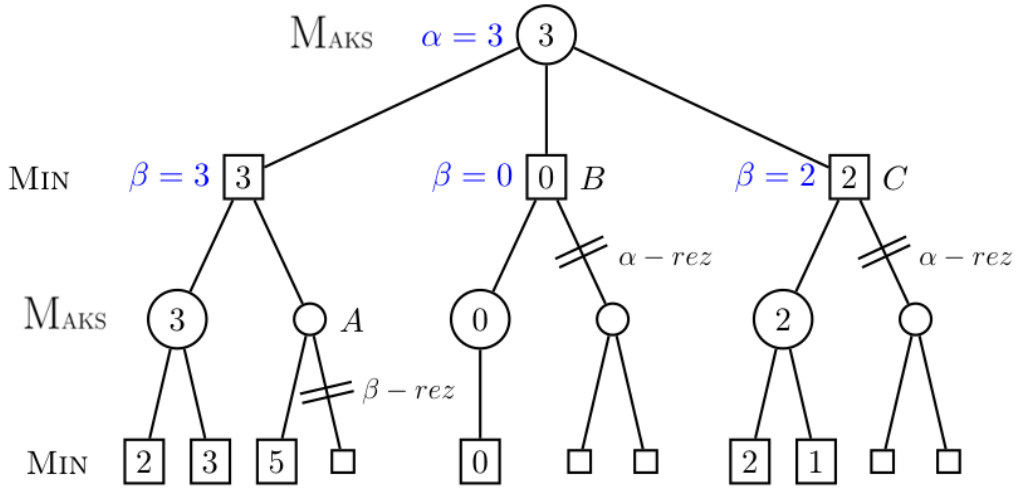
vejitveni faktor drevesa (dovoljeni premiki iz položaja), potem je časovna zahtevnost $O(b^m)$.

3.3 Alfa-beta rezanje

Zaradi eksponentne eksplozije drevesa pri algoritmu minimaks je treba uporabiti nekatere optimizacije. Najpogostejše metode so rezanje neobetavnih vej drevesa. Mi smo pri tem uporabili alfa-beta rezanje.

Algoritem alfa-beta upošteva principe algoritma minimaks, vendar temelji na načelu, da za premike, ki so dovolj dobri, ni treba ugotoviti točno kako dobri so. Prav tako za premike, ki so slabši od že ocenjenih, ni potrebno vedeti natančno, kako slabi so.

Alfa-beta algoritem je dobil ime po parametrih, ki predstavljata meje za vzvratni prenos ocen tekom preiskovalne poti *najprej v globino*. α je vrednost najboljše ocene, ki smo jo našli na poti do sedaj za igralca $MAKS$. β je vrednost najboljše ocene na poti za igralca MIN. Alfa-beta iskanje posodablja vrednosti α in β in odreže preostala poddrevesa, takoj ko ocena njihovega vozlišča pade pod trenutno vrednost α ali β za igralca $MAKS$ ali MIN.



Slika 13.1: Alfa-beta rezanje na drevesu minimaks [23]

Primer na sliki 13.1 prikazuje alfa-beta rezanje. Algoritem najprej sestopi do skrajnega levega vozlišča, si zapomni vrednosti 2 in 3 ter izmed njih izbere največjo vrednost vozlišča $MAKS$ $max(2, 3) = 3$. Izbrano vrednost 3, prenese višje v vozlišče MIN in tam nastavi vrednost $\beta = 3$. Sestopi v vozlišče A, kjer najprej najde vrednost 5. Ker je vrednost 5 večja od vrednosti β v staršu, izvede β-rez vozlišča A in posodobi vrednost α v korenu. Sledi sestop do skrajnega levega vozlišča v B in posodobitev β za vozlišče B z vrednostjo potomca. Ker je to MIN vozlišče, ki je že sedaj nižja od vrednosti α v korenu, je jasno da v ostalih potomcih ni mogoče dobiti višje vrednosti, tako da je mogoč α-rez vozlišče B. Enako sledi v vozlišču C, v β pa se prenese vrednost $MAKS$ potomca, ki je manjša od vrednosti α v vozlišču, zato se vozlišče C lahko α-reže. Iz korakov sledi, da je najboljša vrednost v korenu 3.

3.4 Hevristična ocenjevalna funkcija

Ker je minimaks tudi z alfa-beta rezanjem izredno požrešen algoritem, so le redke igre dovolj preproste, da je mogoče preiskati vse možnosti do zmage ali poraza.

Zato je treba omejiti pregledovanje drevesa na določeno globino in oceniti, ali so plošče na tej globini dobre ali slabe. Ta ocena je bistvena, kako dobro bo igral računalnik. Nenazadnje ni koristno pregledati drevo 20 premikov naprej, če se na tej globini oceni, da je plošča ugodna, ko pa je v resnici porazna. Dobra hevristična ocenjevalna funkcija naj bi ocenila ploščo čim bolj podobno kot prava ocenjevalna funkcija. To pomeni višje vrednosti za plošče, ki vodijo v zmago, okolica ničle za izenačenje in negativne vrednosti za plošče, ki vodijo v poraz. Obvezno je tudi, da je ocenjevanje plošče hitra operacija, saj je njen namen pohitriti samo iskanje najboljšega položaja.

Za sestavo dobre hevristične ocenjevalne funkcije je potrebno dobro poznavanje domene naše igre, zato ne obstaja univerzalni recept, ki bi deloval za vsako igro. Vendar se pri sestavi pogosto da uporabiti osnovni princip *utežne linearne funkcije* [22]:

$$eval(s) = \sum_{i=1}^{n} w_i f_i(s) \tag{3.1}$$

Kjer je s stanje plošče, ki jo ocenjujemo, i so posamezne figure na plošči, w_i je utež, ki jo pripišemo figuri i, f_i pa je njena funkcija na plošči. Za bolj kompleksne igre je slabost *utežne linearne funkcije* v tem, da predpostavlja neodvisnost prispevkov različnih figur. Zato se za izboljšanje ocene pogosto uporablja tudi *nelinearna* kombinacija funkcij.

Na primeru igre Gobblet smo za težo figure w_i izbrali njeno velikost, saj so se empirično večje figure izkazale za veliko močnejše, ker jih manjše ne morejo prekriti in tako držijo svoj položaj na plošči. Njihovo funkcijo f_i smo določili na podlagi vrste, za katero je treba oceniti položaj. Več večjih istih figur v vrsti, je dober napovedovalec, bolj uspešne plošče.

Hevristična ocena plošče za Gobblet

Za Gobblet 3×3 in Gobblet 4×4 smo uporabili isto hevristično ocenjevalno funkcijo. To je bilo možno, ker je najmanjša velikost figure enaka 1, največja

pa je enaka eni dimenziji plošče, torej 3 za 3×3 igro in 4 za 4×4. To funkcijo bi bilo možno uporabiti tudi za ocenjevanje večjih plošč, za večjo hipotetično igro Gobblet, kot recimo 5×5.

Funkcija izbere vrste, po katerih se lahko postavijo tri ali štiri v vrsto (pri igri 4×4). To so vse vertikalne, vse horizontalne ter obe diagonali. Za vsako vrsto posebej se preveri njene figure, nato jih ustrezno ocenimo ter prištejemo k skupni oceni plošče. Oceno vrste sestavlja njena utež figure, ki jo predstavlja velikost figure in pa število figur enega igralca v tej vrsti. Predznak ocene ene vrste določa ali gre za igralca $MAKS$ ali pa za igralca MIN. Igralec $MAKS$ ima pozitiven predznak, igralec MIN pa negativen. Tako po končni izračunani oceni že na podlagi predznaka vidimo, kateri igralec naj bi po naši oceni zmagoval na tej plošči.

Potek ocenjevanja na plošči 3×3 poteka takole: Skupno obstaja 8 smeri, pri katerih se lahko doseže 3 v vrsto. Za vsako od teh vrst je treba preveriti:

- Če obstaja 3 v vrsto za $MAKS$ igralca, pomeni konec igre, ocena je maksimalna pozitivna vrednost.
- Če obstaja 3 v vrsto za MIN igralca, pomeni konec igre, ocena je minimalna negativna vrednost.
- Če so v smeri samo figure za igralca $MAKS$ velja:
  - Za vsako prazno polje v vrsti je ocena 1.
  - Za vsako najmanjšo vrhnjo figuro velikosti 1 na smeri je ocena 10.
  - Za vsako večjo figuro je njena ocena sestavljena iz $i * 10$, kjer je i, njena velikost (1, 2 ali 3).
  - Končna ocena ene smeri je zmnožek ocen vseh treh polj.
- Če so na smeri figure obeh igralcev $MAKS$ in MIN, potem

  - Če so figure igralca MIN, manjše kot razpoložljive figure izven plošče igralca $MAKS$, potem na isti način, kot zgoraj ovrednotimo figure igralca $MAKS$.

  - sicer je ocena vrste 0.

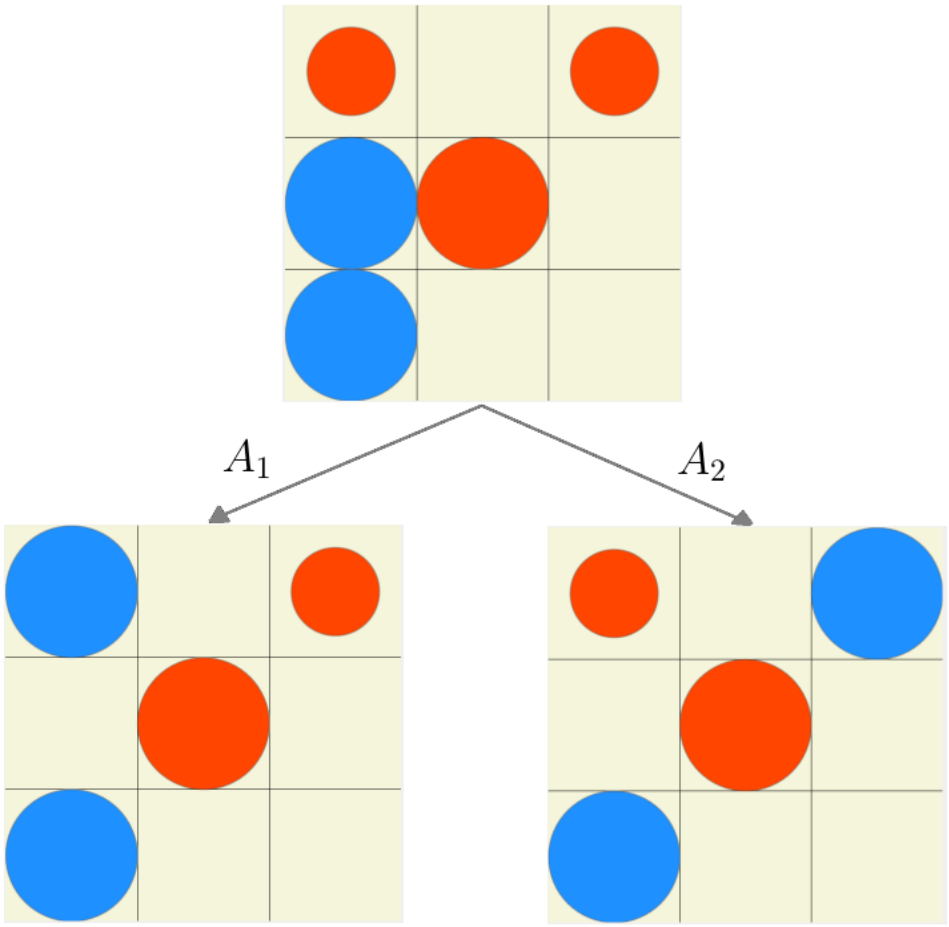
- Za figure igralca MIN igralca se izračuna ocena smeri na enako kot za igralca $MAKS$, vrednost pa je negativna.

- Končna ocena plošče je seštevek vseh smeri.

Določanje hevristične ocene na primeru

Določitev hevristične ocene na primeru igre Gobblet 3×3, ki ga prikazuje slika 17.1. Koren predstavlja stanje plošče pred premikom modrega igralca. Če modri igralec izbere premik A_1, bo naslednje stanje predstavljala leva plošča, v primeru izbire A_2, pa stanje predstavlja desna plošča. Za odločitev kateri premik je boljši za modrega igralca, je treba oceniti vsako ploščo posebej.

Tabela 3.1, prikazuje izračun ocene za ploščo po premiku A_1. Vrstice v tabeli ponazarjajo vse možne smeri, ki pripeljejo do zmage. Vsaka smer se oceni posebej, tako za modrega, kot za rdečega igralca. Ker se išče najboljša možna poteza za modrega igralca, ta predstavlja igralca $MAKS$ in zato zanj velja pozitivna ocena. Rdečemu igralcu se pripiše negativna ocena, po postopku opisanem v poglavju 3.4. Primer 1. smeri vertikalno iz tabele 3.1 za modrega igralca prikazuje izredno visoko oceno, saj ima na tej smeri dve največji figuri. Diagonala v desno pa je za oba igralca ocenjena z 0, saj z največjimi figurami, drug drugemu preprečujeta doseči tri v vrsto. Skupna ocena je seštevek vseh smeri, v tem primeru 850, kar prikazuje premik A_1 kot obetaven. To vidimo tudi iz slike, na kateri ima rdeči zaprti dve možnosti za tri v vrsto in obenem je odprta takšna priložnost modremu.



Slika 17.1: Primer dveh možnih premikov za modrega igralca pri igri Gobblet

Tabela 3.2, prikazuje računanje ocene plošče po premiku A_2. Razvidno je, da je modri s tem premikom sicer zaprl eno možnost za tri v vrsto rdečemu, vendar dva rdeča po diagonali v desno, odpirata možnost za takojšnjo zmago rdečemu v naslednji potezi. To pokaže tudi ocena plošče po premiku A_2, ki je -600. Že zaradi negativne ocene je jasno, da je premik veliko bolj ugoden za nasprotnega igralca.

Algoritem minimaks z nivojem preiskovanja 1, bi pri tem primeru za ustrezen premik izbral višjo oceno izmed 850 in -600, to je 850, do kamor vodi premik A_1.

	Rdeči	Modri
1. vertikalno	0	$(30 \cdot 1 \cdot 30)$
2. vertikalno	$-(1 \cdot 30 \cdot 1)$	0
3. vertikalno	$-(20 \cdot 1 \cdot 1)$	0
1. horizontalno	0	0
2. horizontalno	$-(1 \cdot 30 \cdot 1)$	0
3. horizontalno	0	$(30 \cdot 1 \cdot 1)$
Diag. v desno	0	0
Diag. v levo	0	0
Skupaj za igralca	-80	930
Skupaj ocena	850	

Tabela 3.1: Ocena plošče po premiku A_1

	Rdeči	Modri
1. vertikalno	0	0
2. vertikalno	$-(1 \cdot 30 \cdot 1)$	0
3. vertikalno	0	$(30 \cdot 1 \cdot 1)$
1. horizontalno	0	0
2. horizontalno	$-(1 \cdot 30 \cdot 1)$	0
3. horizontalno	0	$(30 \cdot 1 \cdot 1)$
Diag. v desno	0	0
Diag. v levo	$-(20 \cdot 30 \cdot 1)$	0
Skupaj za igralca	-660	60
Skupaj ocena	-600	

Tabela 3.2: Ocena plošče po premiku A_2

Poglavje 4

Metoda Monte Carlo

V tem poglavju bo predstavljena metoda Monte Carlo, metoda drevesnega preiskovanja Monte Carlo, in uporabo v umetni inteligenci.

4.1 Izvor

Metoda Monte Carlo, iz katere izhaja iskalno drevo Monte Carlo, spada v razred algoritmov, ki za izračun rezultatov uporabljajo naključno generirane razporeditve. Metoda je nastala v znanstvenem laboratoriju Los Alamos leta 1946, ko za pot, ki jo nevtroni prepotujejo skozi različne materiale, niso našli analitične rešitve. Zato sta John Von Neumann in Stanislaw Ulam predlagala, naj poskusijo z računalniškim eksperimentom, v katerem bi generirana naključna števila predstavljala naključna stanja sistemov za katere bi izračunali njihovo vrednost [11]. Navdih za to metodo je Ulam dobil iz igre s kartami. Tajnost vojaškega projekta je zahtevala kodno ime, zato so metodo poimenovali po mestu igralnic Monte Carlu.

Razvitih je bilo veliko različic metode Monte Carlo, vendar v osnovi vse sledijo naslednjim korakom:

1. Definicija domene vhodnih spremenljivk.

2. Generiranje naključnjih vhodnih spremenljivk iz domene na osnovi verjetnostne porazdelitve.

3. Deterministični izračun vrednosti na podlagi vhodnih spremenljivk.

4. Združevanje in interpretacija izračunanih vrednosti.

Uporaba metode Monte Carlo je močno razširjena v različnih naravoslovnih znanostih in ekonomiji. Na področju igranja iger je bila leta 1993 [6] za igranje igre go predlagana kot alternativa algoritmom, ki so odvisni od ocenjevalne funkcije. Opisana metoda [6] je sestavljena iz naslednjih korakov. Za trenutno stanje igre imamo množico dovoljenih premikov. I-krat, kjer je I število ponovitev, naključno izberemo en premik iz množice dovoljenih premikov. Iz vsakega izbranega premika:

1. Pri simulaciji igranja igre, se naključno izbirajo poteze (obeh igralcev), do končnega stanja.

2. V končnem stanju, ko ni možna nobena poteza več, se izračuna vrednost igre (zmaga, poraz, izenačenje).

3. Shranijo se vrednost igre in število simulacij, za izbran premik.

Po koncu ponovitev v vsakem premiku se izračuna povprečno vrednost igre (npr.: št. zmag/št. obiskov vozlišča) in izbere premik z najvišjo povprečno vrednostjo igre.

Po tej metodi se lahko napove verjetnost zmage za oba igralca. Presenetljivo je, da kljub preprostosti metode tako pridobljene napovedi obetajo nekaj točnosti, sploh v igrah, kjer je strateško razmišljanje pomembnejše od taktičnega. V istem članku [6] so predstavili metodo na igri go:

1. Izvajajo se simulacije Monte Carlo igranja, dokler je na voljo dovolj časa. Za vsako simulacijo se posodobi verjetnost zmage za prvi premik.

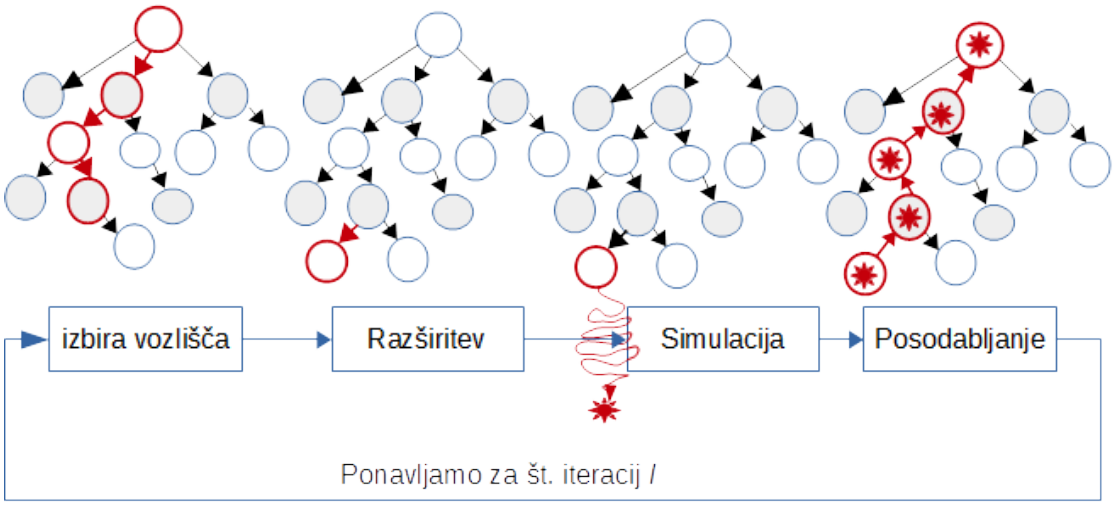
2. Na koncu se izbere premik, ki ima največjo verjetnost zmage.

Ta metoda igranja se je na go različici za ploščo 9x9 izkazala za bolj uspešno kot začetniki igranja go-ja, tudi z zelo malo simulacij na posamezni premik.

Kljub temu, da je bila enostavna simulacija iger po metodi Monte Carlo zanimiva ideja, so implementacije, osnovane izključno na tej ideji, dajale slabe rezultate. Razširitev nad 5000 simulacij ni izboljšala natančnosti. Leta 2006 je Rémi Coulom [10] z ostalimi raziskovalci razširil to idejo v drevesno preiskovanje Monte Carlo in jo uporabil za računalniško igranje igre go. Istega leta sta podoben princip formalizirala tudi Kocsis in Szepesvári [15] v algoritmu UCT.

4.2 Struktura drevesnega preiskovanja Monte Carlo

Leta 2006 je R. Coulom [10] posodobil idejo, da premikom, ki po nekaj simulacijah obetajo boljše rezultate kot ostali, posvetimo več pozornosti (več simulacij igranja). Algoritem postopoma gradi igralno drevo, v katerem pomni rezultate preteklih simulacij in postaja vse bolj natančen pri ocenjevanju najbolj obetavnih potez. Sestavljajo ga štirje koraki, ki jih prikazuje slika 22.1 in se ponavljajo, tolikokrat kot jim določimo v številu iteracij *I*.



Slika 22.1: Osnovni koraki iteracije drevesnega preiskovanja Monte Carlo oz MCTS. Shema je povzeta po [8]

1. **Izbira** vozlišča: Iz korena se rekurzivno izbira optimalnega potomca trenutnega vozlišča, dokler se ne prispe do lista.

2. **Razširitev** vozlišča: Iz izbranega vozlišča se drevo razširi z dodanim novim vozliščem.

3. **Simulacija**: Iz novo dodanega vozlišča se simulira igranje igre do konca.

4. **Posodabljanje** vrednosti vozlišč: Z rezultatom simulacije igre se posodobijo vrednosti vozlišč od novo dodanega vozlišča do korena drevesa.

4.2.1 Izbira vozlišča

Izbiro vozlišča začnemo v korenu drevesa in rekurzivno iščemo najprimernejše razširljivo vozlišče oziroma list. List je vozlišče, ki ni končno (ne predstavlja konca igre), vendar ima neraziskane potomce.

Pri izbiri vozlišča je dobro paziti na uravnoteženje izkoriščanja že znanega in raziskovanje neznanega prostora. Na eni strani izbiramo med najbolj

obetavnimi vozlišči doslej (izkoriščanje). Na drugi pa moramo raziskati tudi manj obetavna vozlišča zaradi možne nezanesljive ocene (raziskovanje). Pri manjšem številu simulacij moramo upoštevati, da vozlišče brez ali s slabo oceno še vedno lahko prinese dober rezultat ob več obiskih.

V svojem članku je Coulom [10] za fazo izbire vozlišča predstavil algoritem *Crazy stone*, ki uporablja formulo, do katere je prišel eksperimentalno. V zaključku je poudaril, da je ta faza nujno potrebna izboljšave. Še istega leta sta Kocsis in Szepesvári [15] predstavila svojo verzijo MCTS, in ga poimenovala strategija zgornje meje zaupanja drevesu, v nadaljevanju UCT (**U**pper **C**onfidence Bounds applied to **T**rees), ki v fazi izbire uporablja formulo zgornje meje zaupanja, v nadaljevanju UCB (**U**pper **C**onfidence Bounds) [14]. Formula UCB se je v fazi izbire izkazala za učinkovito in uspešno metodo za doseganje ravnotežja med raziskovanjem in izkoriščanjem vozlišč drevesa.

Formula UCB nam odloči, katerega potomca k vozlišča p, ki doseže vrednost 4.1:

$$k = \max_{i \in I} \left(\frac{m_i}{n_i} + C \cdot \sqrt{\frac{\ln(n_p)}{n_i}} \right), \tag{4.1}$$

izbrati za prehod na nižji nivo drevesa, kjer je:

- I množica vseh vozlišč dosegljivih iz vozlišča p
- m_i število zmag ali seštevek vseh rezultatov konca iger, v simulaciji igranja iz vozlišča i
- n_p število obiskov vozlišča p
- n_i število obiskov vozlišča i
- C konstanta, ki določa ravnotežje med raziskovanjem in izkoriščanjem.

Konstanta C uravnava razmerje med raziskovanjem in izkoriščanjem drevesa. Z večanjem vrednosti konstante C razvijamo drevo v korist raziskovanja manj obiskanih delov drevesa, nasproti izkoriščanju boljših rezultatov. V

članku [14] je bila predlagana vrednost: $\frac{1}{\sqrt{2}}$, saj le-ta zadosti Hoeffdingovi neeankosti. V praksi se je izkazalo, da je v večini primerov, za ustrezno izbiro konstante C potrebno eksperimentalno določanje.

Če katero vozlišče i še ni bilo nikoli obiskano ($n_i = 0$), je njegova vrednost ∞ in to nam zagotavlja, da bodo vsi potomci vozlišča p dodani v drevo in obiskani vsaj enkrat, preden se bodo potomci razširili naprej.

Vendar samo en obisk vozlišča ni dovolj. Zato večina implementacij izbiri vozlišča doda logiko, ki zagotavlja minimalno število obiskov tega vozlišča, preden začne izbirati vozlišča po formuli 4.1. Dokler ne dosežemo minimalnega števila obiskov vozlišča, se simulacija igranja začenja iz trenutnega vozlišča. Naša izbira je bila konstanta T, ki predstavlja minimalni prag (*threshold*) obiskanosti vozlišča. Prag T smo določili eksperimentalno.

4.2.2 Razširitev vozlišča

V tej fazi algoritem iz izbranega vozlišča, če le-to ni končno (znan je zmagovalec), razširi drevo z novimi vozlišči. Tako se z vsako iteracijo drevo veča, kar posledično pomeni natančnejše preiskovalno drevo.

Najpogostejša strategija razvijanja je dodajanje enega vozlišča za vsako iteracijo. Ta strategija je preprosta in enostavna za implementacijo. Izberejo se lahko tudi druge kompleksnejše strategije, vendar je njihov vpliv na moč algoritma zanemarljiv. V večini primerov zadostuje ustvarjanje enega vozlišča na iteracijo. [7]

4.2.3 Simulacija

Simulacija igre se začne, ko iz na novo dodanega vozlišča izberemo potezo, ki še ni del našega drevesa. Od tu izvajamo izmenične poteze za vsakega igralca, dokler ne pridemo do konca igre. Poteze se lahko izbira naključno ali po simulacijski strategiji. Znano je [13], da vključitev domenskega znanja za razvoj simulacijske strategije, ki je osnovana na hevrističnem znanju specifičnem za posamezno igro, močno izboljša kakovost igranja. Mi smo se

odločili za simulacijo igranja z naključnim izbiranjem premikov.

4.2.4 Posodabljanje

Po koncu simulacije igre rezultat (zmaga, poraz ali izenačenje) shranimo v našem novem vozlišču. Od tu posodobimo vrednosti vozlišč navzgor po drevesu do korena. Vrednost vsakega vozlišča v_i se računa po formuli $\frac{m_i}{n_i}$, kjer pri fazi posodabljanja povečamo m_i za rezultat simulacije in n_i za en obisk.

Vrednost posameznega vozlišča je po koncu vsake iteracije enaka razmerju med seštevkom rezultatov simulacij iger in številom simulacij, ki so vključevale dotično vozlišče.

4.3 Določanje konstant C in T

Za uspešen algoritem drevesnega preiskovanja Monte Carlo (v nadaljevanju MCTS) s strategijo izbiranja UCT sta pomembna pravilna izbira parametrov C in T. Parameter C nadzoruje razmerje med izkoriščanjem uspešnih in raziskovanjem manj raziskanih vozlišč v koraku izbire, medtem ko parameter T določa prag obiskanosti vozlišča. Če je število obiskov vozlišča večje kot prag T, lahko nad vozliščem izvedemo strategijo izbiranja novega vozlišča po UCT, sicer pa se simulacija igranja začne iz trenutnega vozlišča.

4.3.1 Problem slabih konstant C in T

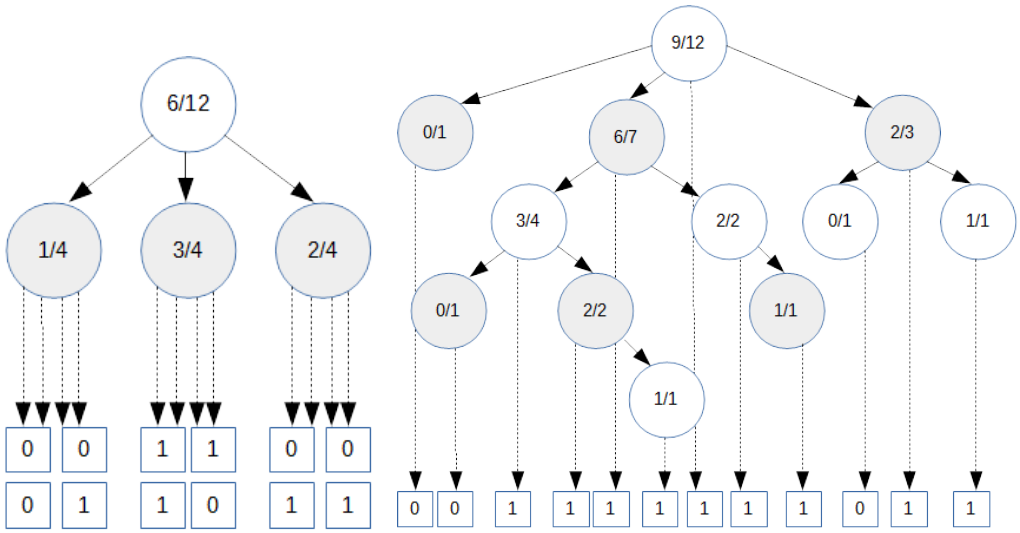
Ob prvi implementaciji MCTS algoritma, smo za parametre C in T izbirali različne vrednosti, ki smo jih našli v literaturi. Rezultati igranja proti takšnemu agentu niso bili obetavni. Odločili smo se za ekperimentalno določitev teh parametrov. Zato smo potrebovali primerljivega nasprotnika. Odločili smo se za implementacijo agenta, ki uporablja samo enostavno Monte Carlo metodo brez drevesnega preiskovanja.

4.3.2 Enostavni Monte Carlo

Enostavni Monte Carlo uporablja metodo opisano v 4.1. Naš bot je iz korena, ki predstavlja začetno ploščo, generiral samo prvi nivo vozlišč, ki predstavljajo igralne plošče, do katerih lahko pridemo s trenutno dovoljenimi potezami za igralca. Iz vsakega vozlišča, je simuliral m iger z naključnimi potezami do zmage ali poraza. V vsakem vozlišču si zapomni število zmag n po m simulacijah. Na koncu izbere potezo, ki vodi do vozlišča z največjo vrednostjo $\frac{n}{m}$.

4.3.3 Namen primerjave MCTS z enostavnim Monte Carlom

Slika 27.1 ilustrira primerjavo med drevesoma, ki ga zgradi enostavni Monte Carlo nasproti metodi MCTS. Enostavni Monte Carlo je zaradi preprostega drevesa izredno hiter, enostaven za implementacijo in prostorsko nepožrešen. Algoritem MCTS, ki gradi drevo, pomni rezultate v vsakem vozlišču posebej, ga rekurzivno preiskuje in z metodo UCT izbira najbolj obetavna vozlišča za začetek simulacije, zato je veliko bolj kompleksen. Implementacija MCTS je popolnoma brez smisla, če nam vrne enake ali celo slabše rezultate kot enostavni Monte Carlo. Zato smo za spodnjo mejo sprejemljivih rezultatov pri določanju parametrov T in C postavili uspešnost, enostavnega Monte Carlo.



Slika 27.1: Levo je primer drevesa, kot bi ga zgradil enostavni Monte Carlo, desno pa drevo, kot bi ga zgradil algoritem MCTS. Povzeto po [12]

4.3.4 Eksperimentalno določanje parametrov

Enostavni metodi Monte Carlo in algoritmu MCTS smo določili skupnega nasprotnika. To je bil algoritem minimaks opisan v poglavju 3.2 s preiskovanjem 3 nivoje globoko (premišljevanje 3 poteze vnaprej). V eksperimentu je bil vsakič kot prvi igralec nasproti algoritmu minimaks enostavni Monte Carlo ali MCTS z različnim naborom parametrov.

Pri MCTS smo za konstanto C testirali naslednje vrednosti :

konstanta C *: 0.001, 0.112, 0.5,* $\frac{1}{\sqrt{2}} \doteq$*0.70711, 1, 5, 25*

Za vrednost praga obiskanosti vozlišč T pa vrednosti:

prag T *: 5, 10, 50, 100, 500*

Za enostavni Monte Carlo smo določili pri vrednosti iteracij za eno vozlišče m=750. Prav tako smo izvedli 100 iger nasproti algoritmu minimaks in dobili verjetnost zmage 0.37. Povprečna vrednost preiskanih končnih iger na eno izvajanje igre je bila 9870. To vrednost smo izbrali tudi za izhodišče pri določanju števila iteracij i algoritma MCTS, ki smo ga nastavili na 10000.

Za vsako vrednost C in T smo izvedli 100 iger proti opisanemu algoritmu minimaks, t.j. skupaj 3500 iger.

Zbrane rezultate predstavlja slika 28.1, iz katere je razvidno, da so dobre vrednosti za prag T vrednosti nad 50, konstanta C pa daje najboljše rezultate okrog vrednosti 0.5.

Eksperiment smo zaradi večje zanesljivosti ponovili z 200 igrami na bolj obetavnih vrednostih iz prvega poizkusa.

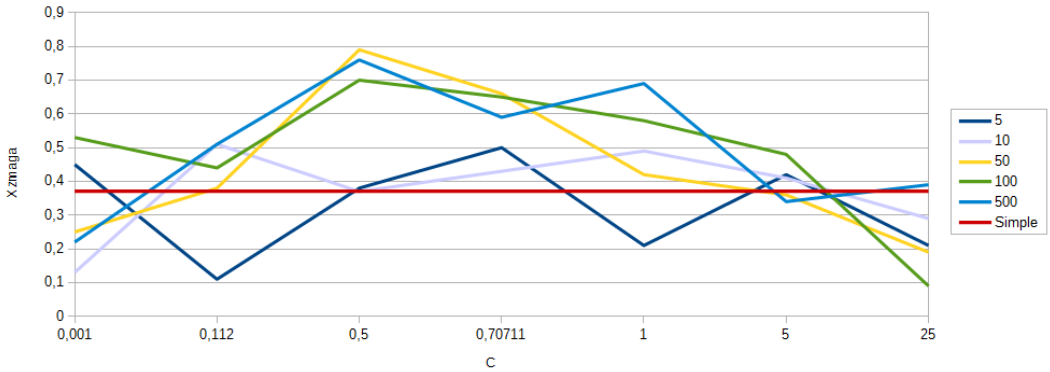
Pri MCTS smo za konstanto C testirali naslednje vrednosti :

***konstanta C** : 0.25, 0.5, 0.75, 1*

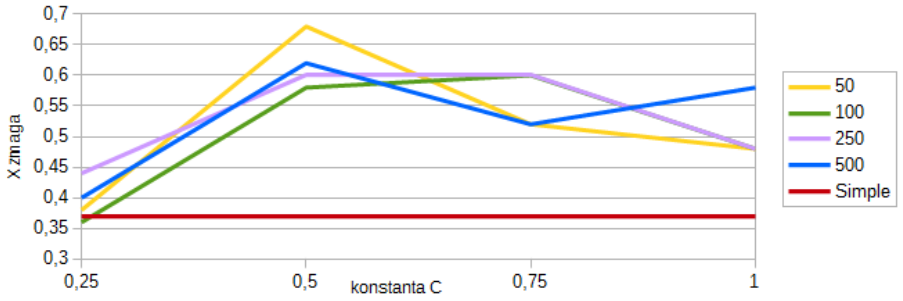
Za vrednost praga obiskanosti vozlišč T pa vrednosti:

***prag T** : 50, 100, 250, 500*

Slika 29.1 predstavlja dobljene rezultate in potrjuje našo izbiro parametrov C=5 in T=50.



Slika 28.1: Delež zmag v odvisnosti od vrednosti konstante C, za vsako vrednost praga T posebej po 100 igrah. Vrednost Simple predstavlja delež zmag za enostavni Monte Carlo.



Slika 29.1: Delež zmag v odvisnosti od vrednosti konstante C, za vsako vrednost praga T posebej po 200 igrah. Vrednost Simple predstavlja delež zmag za enostavni Monte Carlo.

4.4 Predizbira premikov

Slabost MCTS algoritma je, da slabo reagira na možnost neposredne zmage ali poraza. Algoritem MCTS bo izvedel vse določene iteracije izvajanja ne glede na to, kako daleč je do zmage. To je v primerih, ko je za zmago potreben le en premik, časovno potratno in nepotrebno. Zato je dodano preverjanje ali se lahko pride do zmage samo z enim premikom, pred izvajanjem algoritma MCTS. V tem primeru vrnemo zmagovalni premik, ne da bi izvedli vse iteracije MCTS. Prav tako smo upoštevali nasvet Teytauda in Teytauda [24] po katerem se v predizbiri premikov izloči vse premike, ki nasprotniku omogočajo takojšno zmago. To smo izvajali le na prvem vejitvenem nivoju neposredno iz korena. Od tu naprej, če predizbira ni vrnila neposrednih premikov, se je izvajanje MCTS izvajalo normalno.

Poglavje 5

Implementacija igre Gobblet

V tem poglavju predstavimo izdelavo programa za igranje igre Gobblet. Implementirali smo možnost igranja dveh igralcev za računalnikom, enega igralca nasproti agentu, ki igra naključno, agentu z algoritmom minimaks in agentu z algoritmom MCTS. Dodana je tudi možnost opazovanja avtomatske igre dveh enakih ali različnih agentov.

5.1 Okolje in orodja

Zaradi dobrega poznavanja orodja in posledično hitrosti razvoja rešitve smo za implementacijo izbrali Microsoftov programski jezik *C#*. Za razvojno okolje smo uporabili Microsoft Visual Studio 2012. Za vizualizacijo igre smo uporabili grafični programski vmesnik Windows Forms. Uporaba razvitih rešitev je možna v operacijskem sistemu Windows 7 in Windows 8.

5.2 Predstavitev plošče

Predstavitev igralne plošče in njenih pravil smo implementirali z vmesnikom *IBoard*, ki omogoča implementacijo osnovnih funkcij, potrebnih za igranje igre Gobblet.

- *SizeX* in *SizeY* sta dimenzije plošče.

- *GetAvailableFigures*() vrne vse figure, ki še niso postavljene na ploščo.
- *GetFigures*(x, y) vrne figure na polju x, y.
- *AddFigure*$(x, y, newFigure)$, $MoveFigure(fromX, fromY, toX, toY)$ omogoča dodajanje in premikanje figur.
- *GetMovesHistory()* je sklad do sedaj opravljenih premikov.
- $UndoMove()$ omogoči razveljavitev poteze.
- $CellValue$ je razred, ki nam predstavlja posamezno polje in vsebuje definiciji:
  - igralca $Player$
  - velikost figure $Figure$.

Dodatne funkcije za lažje upravljanje s ploščo, kot so vračanje dovoljenih potez *GetAllLegalMovesForPlayer(board, player)* smo implementirali v razredu $BoardExtensions$. Dejanska implementacija plošče se nahaja v razredu $BoardSimple$, ki implementira vse funkcije vmesnika $IBoard$ v skaldu s pravili igre Gobblet.

Zmanjšanje vejitvenega faktorja drevesa

Funkcijo *GetAllLegalMovesForPlayer(board, player)* so uporabljali skoraj vsi agenti za ustvarjanje drevesa. V tem primeru število vrnjenih rezultatov pomeni vejitev drevesa v izbranem vozlišču. Za hitrejše izvajanje algoritmov je bolje, če je to število čim manjše. Upoštevali smo nekaj najbolj osnovnih lastnosti figur in igralne plošče, za zmanjšanje vejitve drevesa. Na primer za ploščo 3×3 obstaja za vsakega igralca na začetku 9 igralnih figur, ki jih lahko postavimo na 9 mest na igralni plošči, to pomeni 81 različnih premikov. Upoštevati je treba, da so 3 različne velikosti figur in da je čisto vseeno, ali izberemo prvo, drugo ali tretjo figuro velikosti 3. Za 3 figure in 9 polj je še vedno 27 možnih premikov, ko v naslednjih potezah vključimo druge

dovoljene premike, to hitro naraste na 35. Če vključimo še simetrijo plošče in po zrcaljenju čez vse 3 osi, na koncu dobimo, za prvo potezo le 3 popolnoma različna polja, kar pomeni vejitveni faktor 9 za prvi premik. Po podobnih optimizacijah, smo prišli do povprečnega vejitvenega faktorja okoli 14, vendar je zelo variiral od igre do igre.

5.3 Predstavitev igralnih agentov

Posamezne algoritme za igranje igre Gobblet smo implementirali v obliki posameznih agentov. Vse agente opisuje vmesnik $IBot$, ki definira funkcijo *GetNextMove(board,player)*. Metoda $GetNextMove$ vrne naslednjo igralno potezo, predstavljeno z razredom $GameMove$, ki vsebuje informacije katero figuro, premaknemo na katero igralno polje.

Za potrebe naše naloge smo implementirali igralne agente: AlphaBetaBot, SimpleMonteBot, MCTSBot in RandomBot.

5.3.1 RandomBot

RandomBot implementira vmesnik $IBot$ in ob klicu funkcije *GetNextMove(board, player)* vrne naključni premik iz množice dovoljenih premikov. Za generiranje naključnega števila uporabljamo psevdo naključni generator Random iz Microsoftove sistemske knjižnice [9]. Za seme se izbere privzeta vrednost, število milisekund od pretečenih od zagona računalnika. *RandomBot* ni bil hud nasprotnik. Namen implementacije je za primerjanje meritev uspešnosti preostalih algoritmov. Od vseh ostalih agentov smo pričakovali, da bodo agenta *RandomBot* premagali v najmanj možnih potezah. To se je tudi zgodilo, zato tega eksplicitno ne omenjamo med rezultati iger.

5.3.2 SimpleMonteBot

SimpleMonteBot ob klicu funkcije *GetNextMove(board,player)* vrne premik, za katerega se je odločil na podlagi enostavnega Monte Carla (glej poglavje 4.3.2).

Agent *SimpleMonteBot* smo implementirali le za potrebe določanje konstante C in pragu T, ki smo ju potrebovali za uporabo $MCTSBot - a$.

5.3.3 AlphaBetaBot

Konstruktor *AlphaBetaBot(depth)* ob kreiranju nastavi število nivojev preiskovanja, ki jih podamo v spremenljivki *depth*. *AlphaBetaBot* implementira vmesnik $IBot$. Funkcija *GetNextMove(board,player)* kliče minimaks algoritem, z alfa-beta rezanjem, ki je prikazan spodaj v kodi kot *GetMinMaxValue*.

```
private int GetMinMaxValue(BoardSimple board, PlayerEnum player, int
    maxDepth, int alpha, int beta,
         out GameMove bestMove)
     {
         bestMove = null;
         if (maxDepth < 1 || board.IsGameFinished() != null)
         {
            return GetScore(board, player);
         }
         if (player == PlayerForMax)
         {
             var playerToMoveNext = changePlayer();
             foreach (var gameMove in
                 board.GetAllLegalMovesForPlayer(player))
             {
                 board.ApplyMove(gameMove);
                 GameMove move;
                 int score = GetMinMaxValue(board, playerToMoveNext,
                     maxDepth - 1, alpha, beta, out move);
                 if (score > alpha)
                 {
                     alpha = score;
                     bestMove = gameMove;
                 }
                 if (alpha >= beta)
                     return alpha;
             }
             return alpha;
         }
         else
         {
             var playerToMoveNext = changePlayer();
             foreach (var gameMove in
                 board.GetAllLegalMovesForPlayer(player))
             {
                 board.ApplyMove(gameMove);
                 GameMove move;
                 int score = GetMinMaxValue(board, playerToMoveNext,
                     maxDepth - 1, alpha, beta, out move);
                 if (score < beta)
```

```
                {
                    beta = score;
                    bestMove = gameMove;
                }
                if (alpha >= beta)
                    return beta;
            }
            return beta;
        }
    }
```

5.3.4 MCTSBot

Konstruktorju *MCTSBot(iterations, C, T)* omejitev preiskovanja določimo v spremeljivki *iterations*, parameter C določa konstanto C, T pa prag (threshold), kolikokrat mora biti vozlišče obiskano, predno se za korak izbire uporabi UCB formula. *GetNextMove(board,player)*, kot je prikazan spodaj, v implementaciji *MCTSBot* za izbiranje najboljše poteze uporablja algoritem drevesnega preiskovanja Monte Carlo (opisan v poglavju 4).

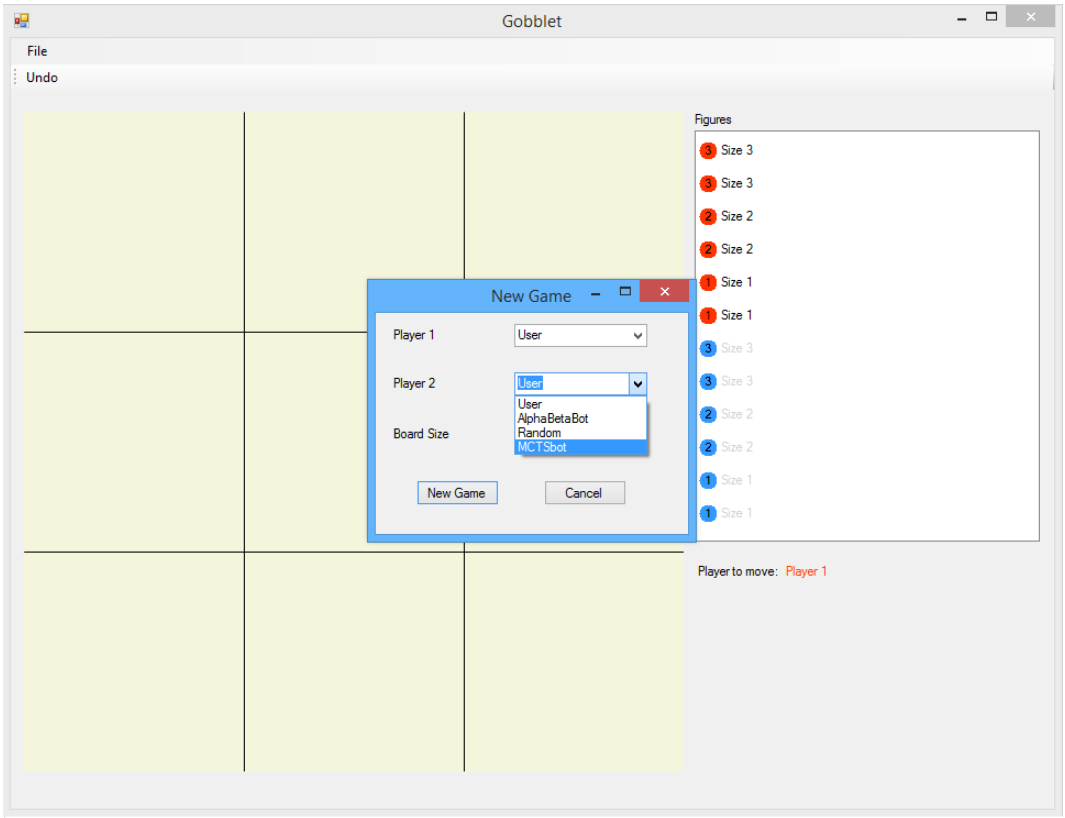
```
public GameMove GetNextMove(BoardSimple board, PlayerEnum playerToMove)
    {
        Node root = new Node(board, null);
        root.playerToMove = playerToMove;
        while (iterations-- > 0)
        {
           Node selected = Select(root);
           selected = Expand(selected);
           Double score = Playout(selected);
           BackPropagate(selected, score);
        }
        GameMove bestMove; double bestValue = -1000;
        foreach (Node node in root.children.Keys)
        {
            if (node.numVisits != 0)
            {
                double value = node.value / node.numVisits;
                if (bestMove == null || bestValue < value)
                {
                    bestValue = value;
                    bestMove = root.children[node];
                }
            }
        }
        return bestMove;
    }
    private Node Select(Node node)
    {
```

```
        if (node.board.IsGameFinished() != null || node.children ==
            null || node.numVisits < threshold)
            return node;
        Node ret; double bestScore;
        foreach (Node childNode in node.children.Keys)
        {
            double score = childNode.value / (childNode.numVisits) + C
                * Math.Sqrt(Math.Log(childNode.parent.numVisits) /
                childNode.numVisits);
            if (ret == null || score > bestScore)
            {
                bestScore = score;
                ret = childNode;
            }
        }
        return Select(ret);
    }
    private Node Expand(Node node)
    {
       BoardSimple boardCopy = node.board;
       var gameMove = Node.movesToExpand[rand.Next(movesCount)];
       node.movesToExpand.Remove(gameMove);
       boardCopy.ApplyMove(gameMove);
       child = new Node(boardCopy, node);
       return child;
    }
    private double Playout(Node node)
    {
        BoardSimple boardCopy = node.board;
        var nextPlayerToMove = node.playerToMove;
        while (boardCopy.IsGameFinished() == null)
        {
          var listOfMoves =
              boardCopy.GameMovesSymmetry(nextPlayerToMove);
            boardCopy.ApplyMove(listOfMoves[rand.Next()]);
            nextPlayerToMove = changePlayer();
        }
        double score;
        if (boardCopy.IsGameFinished() == ourPlayer)
                    score = 1.0;
        else score = 0.0;
        return score;
    }
    private void BackPropagate(Node node, double score)
    {
        node.numVisits++;
        while (node.parent != null)
        {
            node.value += score;
            node = node.parent;
            node.numVisits++;
        }
    }
```

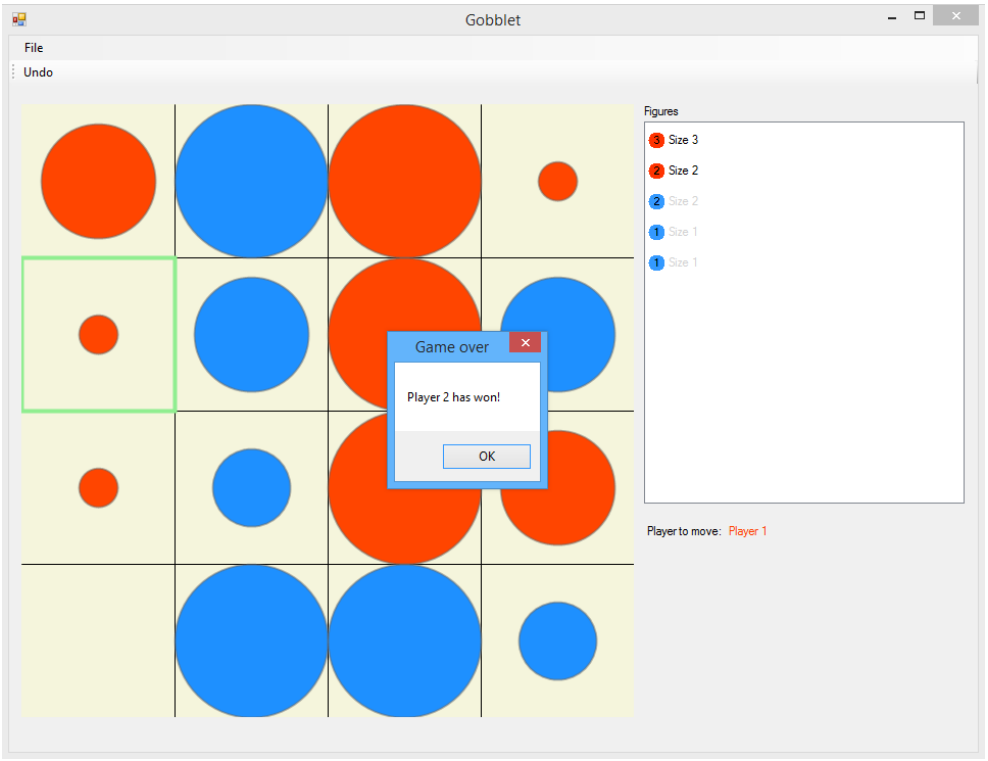
5.4 Predstavitev grafičnega vmesnika

Preprost grafični uporabniški vmesnik za igralce smo izdelali z orodjem Windows Forms. Sestavljata ga osnovna igralna površina in meni za novo igro. V meniju za novo igro (slika 38.1) izberemo, kdo bo prvi in kdo drugi igralec. Izbira se med igralcem za tipkovnico in tremi boti. V meniju pod Board Size izberemo vrsto igre 3×3 ali 4×4. Prvi igralec je vedno rdeči, drugemu je dodeljena modra barva.

Implementacija poteka igre je izvedena tako, da igralca izmenično postavljata ali premikata figure po plošči po principu povleci in spusti. Na začetku so vse razpoložljive figure razporejene v oknu *Figures* desno od igralne plošče. Potek igre tudi opozori igralca na nepravilno potezo in mu prepreči njeno izvršitev. Če je kateri od igralcev agent, izvede potezo samodejno, ko je na vrsti. Slika 39.1 prikazuje konec igre Gobblet na plošči 4×4.



Slika 38.1: Meni za začetek in izbiro nove igre Gobblet



Slika 39.1: Konec Gobblet 4×4, zmagal je modri igralec

Poglavje 6

Rezultati iger

V tem poglavju predstavimo uspešnost implementiranih agentov v igri Gobblet.

6.1 Okolje

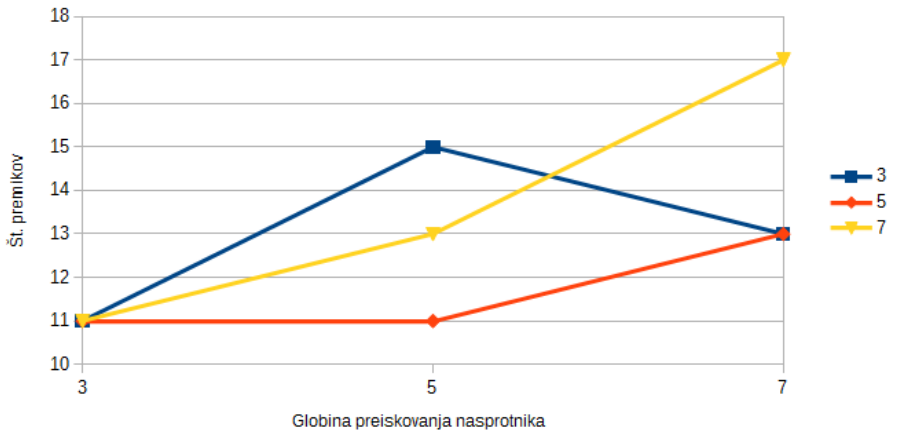
Meritve smo izvajali na računalniku s procesorjem *Intel Core i5-4670, 3.40GHZ* in $16GB$ delovnega pomnilnika. Če ni drugače navedeno, smo rezultate meritev dobili s 100 ponovitvami iger.

6.2 Gobblet 3×3

6.2.1 Agent Minimaks

Pri igri Gobblet 3×3 smo za ocenjevanje uspešnosti agentov upoštevali izkušnjo, da dober igralec, ki začne igro, vedno zmaga. Agent minimaks se je izkazal za tako dobrega, da je bil že 3. nivoja preiskovanja naprej nepremagljiv. Graf na sliki 42.1 prikazuje rezultate iger dveh agentov minimaks. Vsaka linija zase predstavlja agenta minimaks, ki je začel igro in je preiskoval do nivoja, označenega v tabeli. Os x prikazuje do katere globine je preiskoval nasprotnik. Ker je vsakič zmagal začetni igralec, smo ocenjevali število potez, ki jih je prvi igralec potreboval za zmago.

Razvidno je, da je bil za prvega igralca s preiskovanjem do nivoja 3 trd oreh nasprotnik, ki je preiskoval do nivoja 5, presenetljivo pa je lažje premagal nasprotnika, ki je preiskoval vse do globine 7. Prvi igralec na nivoju 5 je z malo truda premagal igralca na nivoju 3 in enako močnega nasprotnika, tudi nivo 7 mu ni predstavljal večjega napora.Prvi igralec na nivoju 7 pa se je obnašal v nasprotju s pričakovanji. Že kot drugi igralec ni predstavljal najmočnejšega nasprotnika, kot prvi pa je sicer solidno zmagal proti nasprotniku na nivoju 3 in nivoju 5, toda proti enako močnemu nasprotniku je potrebovala veliko več potez do zmage, kot so jih potrebovali nasprotniki s preiskovanjem do nižjih nivojev.



Slika 42.1: Graf predstavljaštevilo premikov, ki jih potrebuje agent minimaks kot prvi igralec na določeni globini, da premaga nasprotnika, agenta minimaks na globini, predstavljeni na X osi. Za primerjavo naj omenimo, da je najmanjše število premikov do zmage 5, ob optimalni igri obeh igralcev pa je to število 13.

To presenečenje nam je povzročilo težavo, dokler nismo v literaturi zasledili, da gre za leta 1980 odkriti problem patologije minimaksa [5].

Pojav patologije Minimaksa v rezultatih

Patologija minimaksa je pojav, ko minimaks, povsem v nasprotju z našo intuicijo, pri večji globini preiskovanja vrača slabše rezultate, kot jih je dosegel pri plitkejšem preiskovanju. Z razlago tega pojava so se zadnjih 30 let ukvarjali številni znanstveniki. Najboljšo razlago našega presenečenja smo našli v članku, ki povzame večino raziskav na področju patologije minimaksa: *When Is It Better Not To Look Ahead?* [19].

V članku [19] kot glavne tri parametre, od katerih je odvisna patologija minimaksa, navajajo: vejitveni faktor drevesa b, razdrobljenost vrednosti ocen hevristične ocenjevalne funkcije r in podobnost lokalnih ocen.

Vejitveni faktor drevesa b z večanjem neizogibno vpliva na večjo patologijo. To se zgodi, ker je tendenca algoritma minimaks, da eliminira vse nizke vrednosti na nivoju $MAKS$ in vse visoke na nivoju MIN. Kombinacija povečevanja vrednosti b in sočasno globlje iskanje povzroči večjo verjetnost, da bodo med prehajanjem navzgor odstranjene vse ostale vrednosti razen ene. Posledica tega je, da imajo lahko vsi potomci korena isto vrednost, ne glede na to, kateri premik je najboljši.

Razdrobljenost vrednosti ocen hevristične ocenjevalne funkcije r je velikost razpona vrednosti, ki jih lahko dosežejo ocene plošč po uporabljeni hevristični ocenjevalni funkciji. Nižja razdrobljenost onemogoča natančnejše določanje razlike med položaji, ki so podobni, vendar ne identični. To povzroči večjo možnost, da se z globljim preiskovanjem med prve potomce korena (izmed katerih izbiramo optimalni premik) dobi vse iste vrednosti, kar onemogoči pravilno odločanje kateri je dejansko boljši.

Podobnost lokalnih ocen listov je najbolj raziskan in splošno sprejet vzrok patologije minimaksa, ki se pojavlja se skoraj v skoraj vseh realnih igrah, najti pa je ni v teoretičnih modelih minimaksa. Podobnost lokalnih ocen bomo predstavili na primeru vozlišč a, in b. Če je vozlišče a boljše kot vozlišče b, potem je večja verjetnost, da je večina vrednosti otrok vozlišča a boljših od večine vrednosti otrok vozlišča b (četudi ima b eno samo vrednost ki je večja od vseh v a) in da bi nam globlje preiskovanje to potrdilo.

Pri implementaciji agenta minimaks smo imeli le zelo visoko razdrobljenost ocene hevristične funkcije, kar pa ni zadoščalo, da bi se izognili patologiji minimaksa na globini 7, saj tudi po mnenju ostalih raziskovalcev le ta redko igra glavno vlogo [19]. Po analizi rezultatov za globino 7 in iskanju razlogov za patologijo smo prišli do zaključka, da bi jo teoretično lahko zmanjšali, če bi zmanjšali vejitveni faktor drevesa z vpeljavo bolj natančne funkcije izločanja simetričnih plošč in implementirali, da se navzgor ne bi prenašale samo najvišje vrednosti potomcev, ampak bi se iskalo vozlišče z najvišjo podobnostjo ocen njegovih potomcev. Tega se nismo lotili, saj smo bili zadovoljni z rezultati agentov ki so preiskovali na nivojih 3 in 5. Bi pa morali to idejo upoštevati, če bi želeli izboljšati obstoječega agenta.

Pristranskost hevristične ocene sodih in lihih nivojev

Kot je očitno, sta na grafu 42.1 izvzeti globini preiskovanja 4 in 6. Razlog za tem je znan dobro preučevan pojav osciliranja ocene, ki nam jo vrne minimaks preko hevristične ocenjevalne funkcije v koren za sode in lihe nivoje preiskovanja [17] [18]. Pojav so poimenovali tudi *manično depresivno* obnašanje algoritma, saj je za lihe globine vračal preveč optimistične ocene za sode pa preveč črnoglede. Razlog leži v tem, da smo upoštevali dokaj preprosto hevristično ocenjevalno funkcijo, ki upošteva katere figure imata igralca in kaj pomenijo ti položaji. Na lihih nivojih taka funkcija vrača vrednosti večje od starša ali njej enake, saj se ocena figur po potezi prvega igralca lahko spremeni le njemu v korist, na sodih pa vrača vrednosti manjše od starša, ali njej enake, saj se ocena po potezi drugega igralca lahko spremeni le prvemu v škodo [16].

Teoretično obstajajo načini, da se popravi hevristična ocenjevalna funkcija tako da, ki upošteva ta pojav in vrača čim bolj nivojsko neodvisne ocene. V praksi pa smo se odločili kot večina raziskovalcev, da za končne tekmovalne agente uporabljamo le lihe nivoje.

6.2.2 MCTS proti Minimaks

Kot prvi igralec je bil agent minimaks vsakič nepremagljiv proti agentu MCTS. Zato smo za primerjavo pripravili rezultate, kjer je prvi igralec agent MCTS, s parametri *T,C in I* (kjer je *I* število iteracij), drugi pa minimaks na globinah 3 in 5. Za testiranje smo izvedli 100 iger za vsako kombinacijo nabora parametrov. Rezultati so predstavljeni v tabeli 6.1.

Tabela 6.1: Rezultati za igro Gobblet 3 × 3. Vrednosti prikazujejo delež zmag za prvega igralca X, pri čemer je igralec X: agent MCTS s podanim naborom parametrov, igralec Y: agent minimaks s podanim nivojem globine preiskovanja.

$Y(nivo)$ / $X(C,T,I)$	3	5
0.5, 50, 10000	0.68	0.02
0.5, 200, 30000	0.72	0.09

Očitno je, da agent MCTS ni tako dober igralec kot minimaks, saj bi kot uspešen začetni igralec moral vedno zmagati. Vendar še vedno uspe pogosto premagati agenta minimaks, kadar le ta preiskuje le do nivoja 3. Povečevanje števila iteracij nam je le malo povečalo možnost za zmago. Agent minimaks z nivojem preiskovanja 5 pa se je v večini primerov izkazal za pretežkega nasprotnika.

6.3 Gobblet 4 × 4

Različica Gobblet 4 × 4 je še enkrat daljša kot 3 × 3. Nismo našli očitne strategije, po kateri bi zagotovo zmagal določen igralec, in rezultati v tabeli 6.2 igre dveh agentov minimaks na različnih globinah ne kažejo velike prednosti za prvega igralca.

Tabela 6.2: Rezultati dveh minimaks agentov za igro 4×4. Vrednosti prikazujejo zmage za prvega igralca X, ali drugega igralca igralec Y in vrednost premikov do konca igre. V primeru remi-ja, število premikov pove, kdaj so se premiki začeli ponavljati.

$X(nivo)$ \ $Y(nivo)$	3	4	5
3	X(17)	remi(31)	Y(26)
4	X(23)	remi(60)	Y(46)
5	X(23)	remi(67)	remi(55)

Če oba agenta igrata dobro, se igra hitro konča z remijem. Remi pri igri Gobblet ni eksplicitno definiran v pravilih, vendar prav tako ni eksplicitno prepovedano izvajanje ponavljanja potez do katerega pride, kadar bi vsak drugačen premik razen ponovljenega igralca prisilil v poraz. Tako da smo nad implementacijo Gobbleta dodali odkrivanje ciklov, ki pokaže, kdaj se tretjič ponovi isti vrstni red potez in razglasi končni rezultat remi.

6.3.1 MCTS proti minimaksu

Kljub temu, da se pri igri dveh minimaks agentov ni pokazala prednost prvega igralca, je bila ta očitna, kadar je igro začel agent minimaks na katerekoli globini preiskovanja nasproti MCTS, saj je vsakič zmagal.

V tabeli 6.3 si lahko ogledamo rezultate 100 odigranih iger za vsak set parametrov, ko je bil agent MCTS začetni igralec. Rezultati predstavljajo igre, v katerih smo izklopili odkrivanje ciklov in zato ni prihajalo do remijev.

Tabela 6.3: Rezultati za igro Gobblet 4×4. Vrednosti predstavljajo delež zmag za prvega igralca X, ki je agent MCTS, s podanimi parametri, proti igralcu Y, ki je agent minimaks s podanim nivojem preiskovanja.

$X(C,T,I)$ \ $Y(nivo)$	3	5
0.5, 50, 10000	0.16	0
0.5, 200, 30000	0.34	0.05

Delež zmag agenta MCTS ni bil ravno obetaven. Kot smo že omenili pravila za ukrepanje ob ponavljanju potez niso eksplicitno določena za igro Gobblet. Pri igri dveh agentov minimaks je pojav ciklov očiten, saj vsak agent izračuna isto najboljšo oceno, in od te ocene ne odstopi, če drugačna poteza pomeni gotov poraz. Tako da brez logike za odkrivanje ciklov igra lahko traja neskončno dolgo. Zaradi elementa naključnosti pa agent MCTS na številu iteracij, ki smo mu jih določili redkeje izvede trikrat isto potezo.

Kot zanimivost so v tabeli 6.4 navedeni rezultati igre, v kateri se sproži odkrivanje cikelov že po drugi ponovitvi in razglasi remi. V tem primeru se je agent MCTS izkazal za težjega nasprotnika agentu minimaks.

Tabela 6.4: Vrednosti predstavljajo delež zmag in remijev igralca X, ki je agent MCTS proti igralcu Y, ki je agent minimaks za igro Gobblet 4×4 z detekcijo ciklov, ki pomenijo remi.

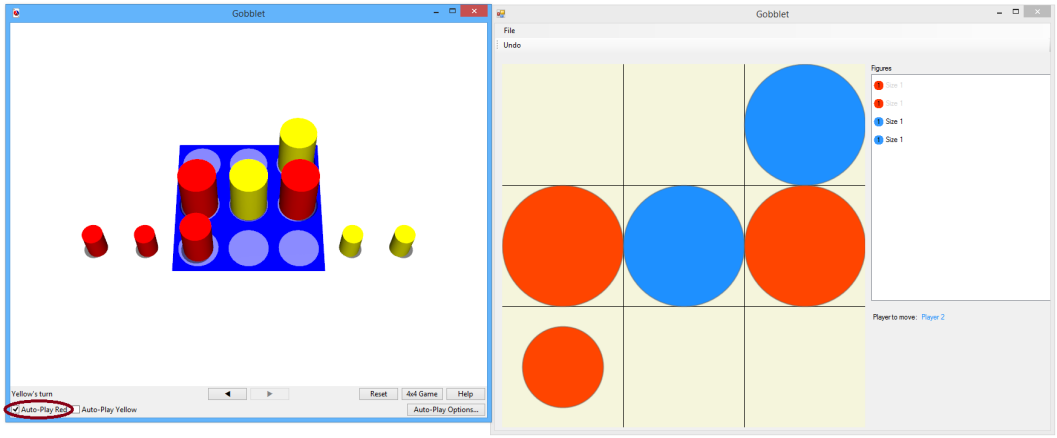
$Y(nivo)$ / $X(C,T,I)$	3		5	
	Zmaga X	Remi	Zmaga X	Remi
0.5, 50, 10000	0.14	0.21	0	0.12
0.5, 200, 30000	0.35	0.34	0.06	0.19

6.4 Primerjava proti Gobblet Racket

Kot zanimivost smo izbrane agente primerjali še z obstoječo razvito igro Gobblet omenjeno v poglavju 2.4. Igro smo izvedli ročno. To pomeni, da smo sočasno zagnali implementacijo Racket in našo verzijo. Pri Racket verziji smo kot prvega ali drugega igralca vklopili *Auto play Red/Yellow*, našo verzijo pa smo nastavili za enega igralca nas, kot drugega pa avtomatsko igro z minimaks agentom ali MCTS agentom. V obeh implementacijah začne rdeči, modri pa ima isto funkcijo drugega igralca kot Racket rumeni igralec. Za prvega igralca v Racket smo nastavili *Auto play Red*, v naši igri pa smo kot prvega igralca nastavili igralca za računalnikom, kot drugega pa vklopili agenta minimaks.

Igro je začel program Racket in postavil prvo rdečo figuro, ki smo jo prav tako postavili na naši plošči, potem smo počakali da je naš agent postavil modro figuro na naši plošči, in nato isto potezo izvedli na sosednji plošči z rumeno figuro, kot predstavlja slika 48.1.

Pri uporabi agenta minimaks se je hitro izkazalo, da se ponavljanje iger zelo hitro spremeni v dve identični množici potez, zaradi same narave algoritma minimaks, ki sta ga v tem primeru uporabljali obe implementaciji.



Slika 48.1: Primer ročne igre proti Gobblet Racket

Za igro 3×3 je agent minimaks - za obe globini 3 in 5 - igral enako močno igro kot Racket. To pomeni, da je igro dobil vsakič, ko je začel, ko pa je igral kot drugi igralec, je zgubil v istem številu potez (13) kot Racket, proti agentu minimaks.

Na plošči 4×4, je naš agent minimaks vedno premagal Racketa, ne glede na to ali je bil začetni igralec, ali pa je igral kot drugi. Zasluge temu pripisujemo naši boljši implementaciji hervistične funkcije.

Prav tako je naš agent MCTS na plošči 4×4 s parametri 30000 iteracij, konstanto $C = 0.5$ in prag $T = 200$, vedno premagal Racket, kadar je začel igro. Kadar pa je igral kot drugi igralec, je bil njegov delež zmag 0.57. Pri

igri z agentom MCTS se zaradi narave naključnosti ni bilo mogoče zanašati, na identične igre, ki smo jih zaznali pri agentih minimaks, zato smo obe igri (kot začetni igralci in kot drugi) izvedli vako ročno s 30 ponovitvami, kar sicer ni dovolj za statistično zanesljivost rezultatov, vseeno pa ustvari neki občutek uspešnosti agenta.

Poglavje 7

Sklepne ugotovitve

V nalogi smo razvili računalniško različico namizne igre Gobblet. Z uporabo algoritma minimaks in algoritma MCTS smo tej igri dodali dva računalniška igralca s katerima se lahko pomeri uporabnik. Na koncu smo izvedli primerjavo uspešnosti teh dveh algoritmov, kjer sta se pomerila kot nasprotnika.

Minimaks se je izkazal za veliko uspešnejšega pri igranju igre Gobblet kot MCTS. To je bilo v skladu s pričakovanji, saj je igra Gobblet šolski primer iger, pri katerih je algoritem minimaks uspešnejši zaradi precej nizkega vejitvenega faktorja drevesa in zaradi ne tako zahtevnega procesa določitve uspešne hevristične funkcije za ocenjevanje plošče.

Vendar se je tudi implementacija agenta MCTS izkazala za obetavno. Agent MCTS je večino iger proti agentu minimaks sicer izgubljal, a je bilo to predvsem zato, ker se je minimaks v tej domeni izkazal za človeku skoraj nepremagljivega nasprotnika. Mi v igri Gobblet 4×4 proti agentu minimaks že na globini 3 skoraj nismo mogli več zmagati in igra ni bila več zabavna. Igranje proti agentu MCTS pri 30000 iteracijah je bilo izredno zahtevno, vendar se ga je dalo z dobro igro tudi premagati. Za začetnike bi bilo dobro znižati iteracije na 10000, da ne bi takoj obupali, a navkljub temu še vedno imeli močnega nasprotnika.

Kljub uspešni implementaciji in uporabi algoritmov na igri Gobblet je ostalo še veliko možnosti za izboljšave in nadaljne delo. Med njimi je gotovo

upoštevanje podobnosti lokalnih ocen za odpravo patologije v minimaksu. Dodatno optimizacijo bi dosegli z več uporabe simetrije plošče za zmanjšanje vejitvenega faktorja drevesa, shranjevanjem drevesa in optimizirano predstavitvijo plošče za hitrejše preiskovanje ter računanje ocen. Tudi za izboljšanje algoritma MCTS je veliko prostora: lahko bi namesto naključnjega izbiranja vozlišč na podlagi domenskega znanja razvili simulacijsko strategijo ali pa implementirali pohitritev s paralelnim izvajanjem algoritma. Zanimive pa so tudi raziskave na področju hibridnih algoritmov, ki za igranje igre uporabljajo tako principe algoritma minimaks kot MCTS [4]. Za samo popularnost igre pa bi največ doprinesla razširitev implementacije igre Gobblet kot spletno ali mobilno aplikacijo.

Literatura

[1] Gobblet — strategy game. `http://docs.racket-lang.org/games/gobblet.html`. [Online; dostop 22-2-2014].

[2] Opis produkta: Gigamic - gobblet kid na strani amazon.com. `http://www.amazon.com/dp/B0009H9SU2`. [Online; dostop 22-2-2014].

[3] Opis produkta: Gobblet na strani amazon.com. `http://www.amazon.com/dp/B00006L50P/`. [Online; dostop 23-2-2014].

[4] H. Baier and M.H.M. Winands. Monte-carlo tree search and minimax hybrids. In *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pages 1–8, Aug 2013.

[5] D. Beal. An Analysis of Minimax. In M. R. B. Clarke, editor, *Advances in Computer Chess 2*. University Press, Edinburgh, 1980.

[6] Bernd Brügmann. Monte Carlo Go. Technical report, Max-Planke-Inst. Phys., Munich, 1993.

[7] Guillaume Maurice Jean-Bernard Chaslot. *Monte-Carlo Tree Search*. Ph.d. dissertation, Maastricht Univ., Netherlands, 2010.

[8] Guillaume Maurice Jean-Bernard Chaslot, Sander Bakkes, István Szita, and Pieter Spronck. Monte-Carlo Tree Search: A New Framework for Game AI. In *Proc. Artif. Intell. Interact. Digital Entert. Conf.*, pages 216–217, Stanford Univ., California, 2008.

[9] Microsoft Corporation. Library: Random class. `http://msdn.microsoft.com/en-us/library/system.random.aspx/`. [Online; dostop 9-2-2014].

[10] Rémi Coulom. Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In *Proc. 5th Int. Conf. Comput. and Games*, pages 72–83, Turin, Italy, 2006.

[11] Roger Eckhardt. Stan Ulam, John von Neumann, and the Monte Carlo method. 15 (Special Issue, Stanisław Ulam 1909–1984):131–141, 1987.

[12] Sylvain Gelly, Levente Kocsis, Marc Schoenauer, Michèle Sebag, David Silver, Csaba Szepesvári, and Olivier Teytaud. The grand challenge of computer go: Monte carlo tree search and extensions. *Commun. ACM*, 55(3):106–113, March 2012.

[13] Sylvain Gelly and David Silver. Combining Online and Offline Knowledge in UCT. In *Proc. 24th Annu. Int. Conf. Mach. Learn.*, pages 273–280, Corvalis, Oregon, 2007. ACM.

[14] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[15] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved Monte-Carlo Search. Technical Report 1, Univ. Tartu, Estonia, 2006.

[16] Mitja Luštrek. *Patologija v hevrističnih preiskovalnih algoritmih*. Ph.d. dissertation, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, 2007.

[17] Dana S. Nau. *Quality of Decision Versus Depth of Search on Game Trees*. PhD thesis, Durham, NC, USA, 1979. AAI8003633.

[18] Dana S. Nau. The last player theorem. *Artif. Intell.*, 18(1):53–65, 1982.

[19] Dana S. Nau, Mitja Lustrek, Austin Parker, Ivan Bratko, and Matjaz Gams. When is it better not to look ahead? *Artif. Intell.*, 174(16-17):1323–1338, 2010.

[20] J. von Neumann. Zur theorie der gesellschaftsspiele. *Mathematische Annalen*, 100:295–320, 1928.

[21] Peter A. Piccione. In Search of the Meaning of Senet. *Archaeology*, July/August:55–58, 1980.

[22] Stuart J. Russell and Peter Norvig. Artificial intelligence: A modern approach. pages 161–201, 2010.

[23] Gabriel Synnaeve. Bayesian programming and learning for multi-player video games. *PhD. Thesis*, 2012.

[24] Fabien Teytaud and Olivier Teytaud. On the huge benefit of decisive moves in monte-carlo tree search algorithms. In Georgios N. Yannakakis and Julian Togelius, editors, *CIG*, pages 359–364. IEEE, 2010.