Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Neža Belej

Primerjava in optimizacija preiskovalnih algoritmov na primeru abstraktne igre Yinsh

magistrsko delo

magistrski program druge stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Polona Oblak

Ljubljana, 2017

Zahvala

Ob koncu študija naj omenim ljudi, ki so vsak na svoj način pripomogli k temu, da so moja študentska leta bila čudovita in nepozabna. Zahvaljujem se svoji mentorici, izr. prof. dr. Poloni Oblak. Za prvovrstno mentorstvo pri diplomskem in magistrskem delu, pa tudi za pomoč tekom celotnega magistrskega študija. Hvala za pogovore in vse nasvete. Strokovne in življenjske.

Zahvaljujem se svoji družini: staršem, sestri Barbari, moji najboljši prijateljici; ter dragima bratoma Primožu in Petru. Mojima sončkoma Vitu in Jaki, ker mi vedno polepšata dan.

Hvala prijateljem, ki ste poskrbeli, da so v napornih študentskih letih prevladovali predvsem lepi trenutki: Manca, Blaž, Klementina, Alen, Matevž, Veronika, Matej, Jan, Dejan.

Nejc, težko je opisati, koliko mi pomeni tvoja podpora. Hvala, ker si bil ves čas ob meni. V dobrem in slabem.

Neža Belej, 2017

"Sometimes you'll have the impression that there is nothing but chaos on the board. You line up a few of your pieces - and the next thing you notice is that they have changed color! The more pieces on the board, the more difficult it becomes to predict what will happen. But also the more opportunities you have! So don't worry: you'll get your chance. Just stay alert, and you'll create your own little bit of order in the chaos!"

Contents

Seznam uporabljenih kratic

kratica	angleško	slovensko
DFS	depth-first search	preiskovanje v globino
MCTS	Monte Carlo Tree Search	drevesno preiskovanje Monte Carlo
UCT	Upper Confidence bounds for Trees	zgornja meja zaupanja za drevesa
AI	Artificial Intelligence	umetna inteligenca

Povzetek

Naslov: Primerjava in optimizacija preiskovalnih algoritmov na primeru abstraktne igre Yinsh

V magistrskem delu se ukvarjamo s preiskovalnimi algoritmi, s katerimi je možno poiskati rešitve za abstraktne namizne igre. Izberemo si abstraktno igro Yinsh, ki je del znanega projekta Gipf. Cilj magistrskega dela je najti in implementirati pametnega igralca igre Yinsh, ki je zmožen premagati že obstoječe implementacije pametnih igralcev, hkrati pa je konkurenčen v igri proti izkušenemu človeškemu igralcu. Kot osnovno metodo si izberemo algoritem Minimax, za katerega sestavimo več evalvacijskih funkcij, pri čemer se te nadgrajujejo. Na tem mestu naredimo tudi analizo navzočih konstant v ocenjevalni funkciji in poiščemo optimalno kombinacijo le-teh. Osnoven Minimax zaradi velikega vejitvenega faktorja drevesa igre Yinsh ni zadovoljiva rešitev. Glede na naravo igre predlagamo več optimizacij metode Minimax, jih implementiramo, testiramo in evalviramo. Implementiramo tudi algoritem drevesnega preiskovanja Monte Carlo, katerega uspešnost primerjamo z algoritmom Minimax. Na podlagi raziskav in razvitih algoritmov sestavimo končno rešitev. Predlagamo tudi možne izboljšave in nadaljnje delo. Naša končna rešitev prinaša zelo dobre rezultate: prepričljivo premaga večino obstoječih implementacij, zmožna pa je premagati tudi zelo izkušenega človeškega igralca.

Ključne besede

umetna inteligenca, preiskovalni algoritmi, projekt Gipf, Yinsh, Minimax

Abstract

Title: Comparison and optimisation of search algorithms exemplified by abstract game Yinsh

In the thesis we analyse search algorithms that are able to find solutions for abstract board games. We choose the game Yinsh, one of the well known project Gipf games. The goal of the thesis is to find and implement a Yinsh-playing program that is able to defeat all the available computer programs and is strong against experienced human players. As a base method we choose Minimax, for which we implement various gradually upgrading evaluation functions. We analyse the constants in those functions and find the best combinations of them empirically. Because Yinsh has a large branching factor, basic Minimax is not an optimal solution. Based on the nature of the game Yinsh we propose multiple optimisations of the Minimax method, which we implement, test and evaluate. We also implement Monte-Carlo Tree Search method for the game Yinsh and compare its performance to the Minimax based algorithms. Based on the analysed approaches we compose the final solution. We also propose improvements and future work. Our final solution produces very good results. It defeats multiple computer programs and is also strong against experienced human players.

Keywords

artificial intelligence, search algorithms, project Gipf, Yinsh, Minimax

Poglavje 1

Uvod

Projekt GIPF obsega šest abstraktnih namiznih iger izdelovalca Krisa Burma. Gre za tip iger s popolno informacijo, namenjenih dvema igralcema. Projekt predstavlja velik izziv na področju umetne inteligence, saj je rešljivost vsebovanih iger po mnenju Heula in Rothkrantza kompleksen problem [7]. Igre projekta Gipf se igrajo s preprostimi figurami, ki se polagajo na šesterokotno igralno ploščo. Namen projekta je sestaviti ogrodje različnih iger, pri čemer ima vsaka igra nekoliko drugačna pravila in figure (potenciale), s katerimi je možno sestavljati nove tipe iger [17]. Na spletnem portalu Board Game Geek [19] so v sklopu najboljših abstraktnih iger bile igre projekta Gipf ocenjene zelo dobro: igra Yinsh je trenutno na drugem mestu, na četrtem je Tzaar, na petem Dvonn. Ostale (Zèrtz, Gipf, Pünct) so uvrščene pod 45. mestom.

V okviru projekta Gipf obstaja že nekaj raziskav. Največ gradiva obstaja za igro Dvonn: tako se je Mauss v svojem diplomskem delu [12] ukvarjal z implementacijo drevesnega preiskovanja Monte-Carlo, Kilgo pa je v svoji raziskavi [9] naredil napredek v implementaciji algoritma Minimax za igro Dvonn. Dvonner [21] in Holtz [22] sta napredni implementaciji pametnega igralca igre Dvonn. Waltz [16] se v omenjenem članku opisuje kot močni program za igro Tzaar, pri čemer uporablja napredni različici preiskovalnih metod *rezanja dreves Alfa-Beta* in *preiskovanja z dokaznimi mejami (angl. proof-number search).* V svoji doktorski disertaciji je uspešno implementacijo

pametnega igralca igre Gipf ustvaril Wentink [17]. Prav vse igre projekta Gipf pa imajo implementacijo na spletnem portalu Board Space [20], kjer so pametni igralci iger projekta Gipf opisani kot premagljivi predvsem za izkušenejše igralce. Edina izjema tukaj je Dvonn, katerega zadnje različice robotov na portalu predstavljajo izziv tudi izkušenim igralcem.

Problem, ki ga raziskujemo, je iskanje najboljšega pristopa za implementacijo pametnega igralca igre Yinsh glede na njeno kompleksnost. Pri tem nas bo zanimalo tudi, kako bi lahko znanja in strategije izkušenih igralcev integrirali v našo implementacijo. Našo implementacijo želimo tudi testirati, in sicer tako na obstoječih implementacijah kot proti človeškemu igralcu. Dela, ki nam predstavljajo največji izziv, so:

- implementacija Davida Petra [23], ki je svojo odprtokodno implementacijo pametnega igralca igre Yinsh prenesel tudi na spletno različico igre,
- pametni igralci (*dumbot, weakbot, smartbot*) na znanem portalu Board Space,
- implementacija Thomasa Debraya, ki je s svojim Yinsh botom na turnirju Yinsh botov dosegel tretje mesto.

V članku *Solving games: Dependence of applicable solving procedures* [7] se avtorja med drugim navežeta tudi na rešljivost in teoretično vrednost iger projekta Gipf. Yinsh opredelita kot najbolj kompleksno igro v smislu rešljivosti. Yinsh ima že milijone možnih začetnih pozicij: za začetno postavitev obročkov obstaja 7.9×10^{14} možnih pozicij. Tudi ob upoštevanju simetrije igralne plošče se to število znatno ne zmanjša.

Struktura magistrskega dela je sledeča: v poglavju *Yinsh* bomo podrobno opisali igro Yinsh in njena pravila. V nadaljevanju (poglavje 3) sledi opis vseh uporabljenih metod in algoritmov, ki smo jih tekom izdelave magistrskega dela srečali in implementirali. Opisali bomo tudi, kako smo te metode aplicirali na igro Yinsh. V poglavju *Optimizacija* bomo opisali metode, ki

smo jih implementirali z namenom pohitrititi čas razmišljanja našega pametnega igralca. Sledi opis implementacije (poglavje 5), kjer razložimo našo izbiro orodij za izdelavo programa ter opišemo povezavo med strežniškim delom in uporabniškim vmesnikom igre. *Rezultati* nam bodo dali vpogled v konkurenčnost našega programa – opisali bomo, kako se je naš pametni igralec odrezal proti že obstoječim pametnim igralcem ter tudi proti manj in bolj izkušenim človeškim igralcem Yinsha. Obstoječe pametne igralce bomo tudi opisali. V poglavju *Sklep* bomo rezultate ovrednotili in navedli možnosti za nadaljnje delo.

Poglavje 2

Yinsh

Pravila igre so povzeta po uradnih pravilih igre Yinsh [27].

2.1 Sestavni deli

Potrebni sestavni deli za igranje so:

- igralna plošča,
- 5 belih in 5 črnih obročkov,
- 51 žetonov (na eni strani črni, na drugi beli).

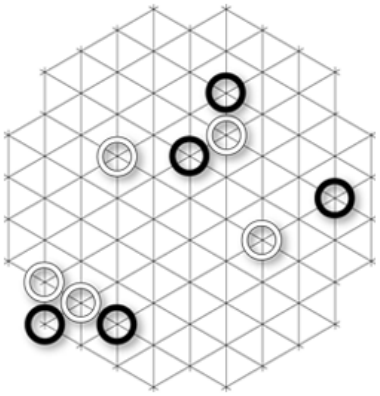
2.2 Cilj igre

Igralca začneta igro vsak s svojimi petimi obročki, postavljenimi na igralni plošči. Igralec sme odstraniti obroček iz igralne plošče, če doseže pet žetonov svoje barve v vrsti. Zmagovalec je tisti, ki prvi odstrani tri svoje obročke iz igralne plošče. Za zmago je torej potrebno nabrati tri vrste po pet žetonov svoje barve.

2.3 Potek igre

2.3.1 Postavitev

Igro začne igralec z belimi obročki. Igralca izmenično postavljata svoje obročke na presečišča šestkotne igralne površine.

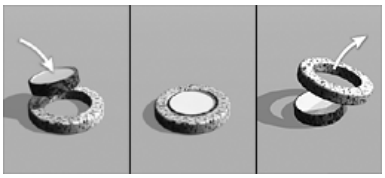


Slika 2.1: Primer stanja igralne plošče po postavitvi obročkov. Vir slike: http://www.boardspace.net/yinsh/english/rules.htm.

2.3.2 Prestavljanje obročkov

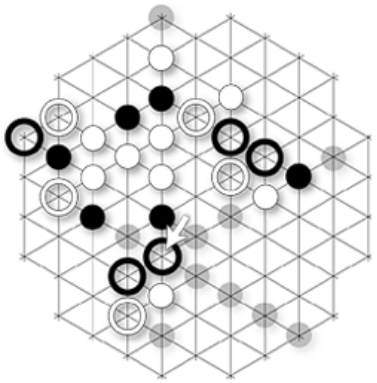
Vsaka poteza se začne s postavitvijo žetona svoje barve v obroček svoje barve. Ta obroček nato igralec premakne po naslednjih pravilih:

- Ob premiku obročka žeton ostane na začetni poziciji obročka (slika 2.2).



Slika 2.2: Premik obročka. Žeton ostane na prejšnji poziciji obročka, obroček se premakne. Vir slike:
http://www.boardspace.net/yinsh/english/rules.htm.

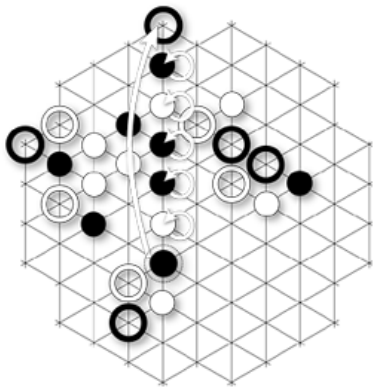
- Obroček se vedno premakne v ravni liniji in na prosto mesto plošče.
- Obroček lahko preskoči eno ali več prostih mest.
- Obroček lahko preskoči enega ali več žetonov poljubne barve, pri čemer morajo biti ti postavljeni zaporedno, tj. brez prostih mest v liniji. Če obroček preskoči žetone, mora obstati na prvem prostem mestu za žetoni v liniji.
- Obroček lahko preskoči najprej prosta mesta in nadaljuje s preskokom čez žetone, a mora po zadnjem žetonu v liniji skok končati.
- Obroček ne sme preskakovati drugih obročkov.



Slika 2.3: Slika prikazuje dovoljene premike označenega črnega obročka. Vir slike: http://www.boardspace.net/yinsh/english/rules.htm.

2.3.3 Obračanje žetonov

Če obroček med potezo preskoči žetone, se vsi preskočeni žetoni obrnejo (spremenijo barvo). To ne velja za žeton, ki je ob začetku poteze položen v premikajoči obroček, saj ni bil preskočen.



Slika 2.4: Slika prikazuje premik označenega črnega obročka. Označeni so žetoni, ki po potezi spremenijo barvo. Vir slike: http://www.boardspace.net/yinsh/english/rules.htm.

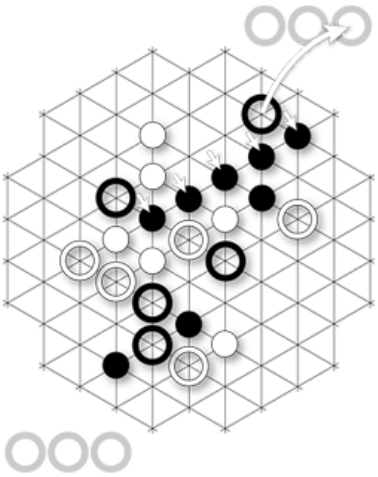
2.3.4 Formiranje vrste in odstranitev obročka

V okviru igre je cilj igralca, da formira vrsto zaporednih petih žetonov svoje barve. Ob tem dosežku igralec vrsto petih žetonov odstrani z igralne površine, prav tako odstrani poljuben obroček svoje barve.

Če igralec doseže več kot pet žetonov v vrsti, lahko odstrani poljubno peterico zaporednih žetonov.

Ob igranju je možno, da igralec z eno potezo pridobi dve vrsti po pet žetonov. Če se vrsti ne sekata, igralec odstrani obe vrsti, prav tako odstrani dva obročka. Če se vrsti sekata, igralec odstrani poljubno vrsto (druga bo po odstranitvi prve nepopolna).

Igralec lahko s svojim premikom formira vrsto petih žetonov v nasprotnikovi barvi. V tem primeru nasprotnik odstrani žetone in obroček in nadaljuje s svojo potezo.



Slika 2.5: Slika prikazuje formirano vrsto petih črnih žetonov.Vir slike: http://www.boardspace.net/yinsh/english/rules.htm.

2.3.5 Konec igre

Igre je konec v naslednjih primerih:

- Igralec pridobi 3 obročke in tako zmaga.
- Igralec z zadnjo potezo pridobi svojo in nasprotnikovo tretjo vrsto. V tem primeru zmaga igralec, ki je odigral potezo, saj je bila njegova vrsta odstranjena prej.
- Če so bili vsi žetoni položeni na igralno ploščo pred formiranjem treh vrst igralca, zmaga igralec z večjim številom pridobljenih obročkov. Če sta nabrala enako število obročkov, je izid neodločen.

2.3.6 Karakteristike igre

Igra Yinsh ima nekaj posebnih značilnosti, ki nam v iskanju njene programske rešitve predstavljajo največji izziv:

- Za razliko od iger tipa *n-v-vrsto* je pri igri Yinsh potrebno zbrati kar tri vrste po pet žetonov. Potrebno je določiti ustrezno ravnotežje med zmago in eno pobrano vrsto petih žetonov.

- Začetna postavitev je dinamična. Igralca izmenično postavljata obročke na poljubno mesto igralne plošče. Za razliko od iger kot sta na primer dama in šah, je v splošnem pri igri Yinsh začetna postavitev pri vsaki igri drugačna. To igralcu onemogoča uporabo vnaprej določenih začetnih potez.

- Igra vsebuje obračanje žetonov, kar pomeni, da se lahko stanje igre v trenutku popolnoma spremeni.

Poglavje 3

Preiskovalni algoritmi

3.1 Minimax

Minimax je najpogostejši algoritem za določanje najboljše poteze iz trenutnega stanja igre. Je zelo uporaben pri enostavnih igrah za dva igralca, kjer igralca igrata drug proti drugemu, pri čemer so vse naslednje možne poteze znane obema igralcema (tj. igra s popolno informacijo).

Minimax temelji na gradnji *preiskovalnega drevesa (angl. search tree)*, kjer vsako *vozlišče (angl. node)* predstavlja možno potezo. Za vsako vozlišče je možno izračunati vrednost, ki ocenjuje *moč (angl. goodness)* poteze. Glede na te vrednosti se lahko izbere najboljšo možno potezo, ki sledi trenutnemu stanju igre. Pri tem je potrebno paziti na alternirajočo naravo iger za dva igralca (angl. two-player games), saj vsak igralec izbira potezo, ki je najboljša zanj (in pomeni izgubo za nasprotnika).

Minimax lahko uporablja dve strategiji: prva je preiskano celotno preiskovalno drevo, vse do listov (angl. leaf nodes), ki ponazarjajo končno stanje igre (zmaga ali poraz). Druga strategija temelji na omejitvi preiskovalnega drevesa glede na vnaprej določeno globino.

Minimax je algoritem, ki uporablja preiskovanje v globino (angl. depth-first search) in pri tem vzdržuje najmanjšo ali največjo vrednost izmed vozlišč naslednikov danega vozlišča. Ob dosegu končnega vozlišča (lista) se izračuna

vrednost tega vozlišča s pomočjo evalvacijske funkcije. To vrednost propagiramo po drevesu navzgor, vse do korena. Na nivojih, kjer je na vrsti naš pametni igralec, se izbere maksimum izmed vrednosti vozlišč naslednikov; za nasprotnika se vzame minimum. Propagiranje po drevesu navzgor je torej alternirajoče: maksimiziramo minimum in minimiziramo maksimum. Z drugimi besedami, privzamemo, da bo vsak igralec odigral potezo, ki je zanj najbolj obetavna [8]. Delovanje algoritma je prikazano na sliki 3.1.

Razložimo še časovno zahtevnost algoritma Minimax. Če je maksimalna globina drevesa m, iz vsakega vozlišča pa je možnih b naslednjih potez, potem je časovna zahtevnost algoritma $O(b^m)$. Prostorska zahtevnost znaša $O(bm)$ za algoritem, ki generira vse akcije naenkrat, ali $O(m)$ za algoritem, ki generira le eno akcijo naenkrat. Preiskovanje v globino namreč zahteva le shranjevanje trenutne poti od korena do lista, skupaj s preostalimi nerazširjenimi sosedi vseh vozlišč na poti. Ko je vozlišče razširjeno, je lahko odstranjeno iz spomina takoj, ko so vsi njegovi potomci že raziskani.

Za realne igre so takšne časovne zahtevnosti popolnoma nepraktične, a algoritem služi kot osnova za matematično analizo iger in za bolj praktične algoritme [15].

Psevdokoda 1 prikazuje splošen Minimax algoritem, povzet po [24].

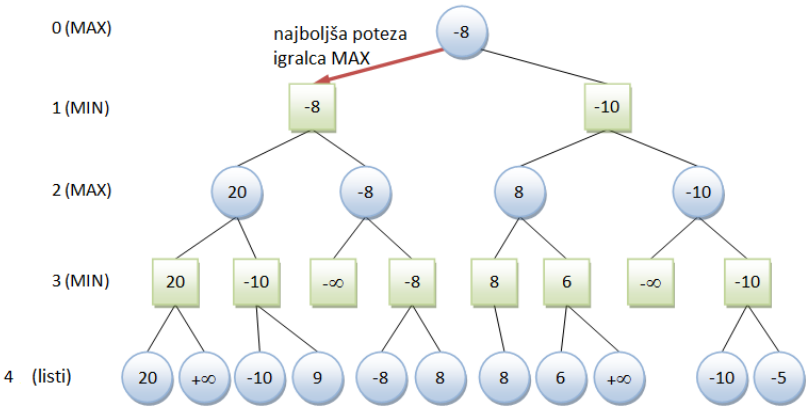
3.2 Rezanje alfa-beta

Glavni problem pri algoritmu Minimax je to, da število preiskovanih vozlišč narašča eksponentno z globino drevesa. Žal eksponentnosti preiskovanja ne moremo odstraniti, a obstaja način, kako eksponent učinkovito prepoloviti. Možno je namreč dobiti točno Minimax vrednost brez da bi preiskali vsako vozlišče drevesa. Algoritem, ki to omogoča, je **rezanje alfa-beta**. Omogoča pridobitev točne Minimax vrednosti, a hkrati zavrže preiskovanje vej, ki ne morejo vplivati na to vrednost [15].

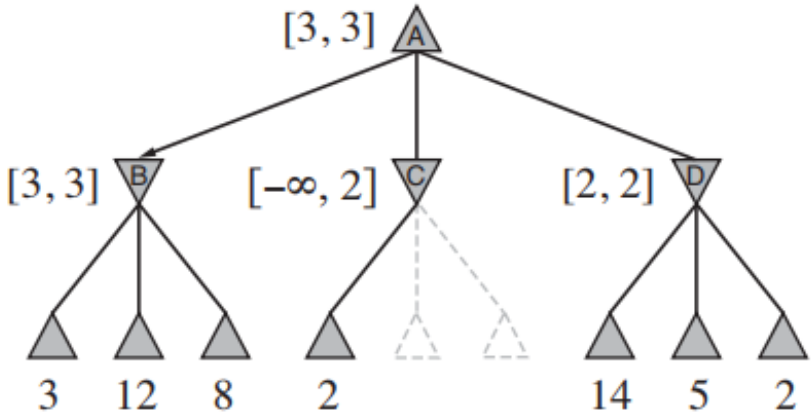
Poglejmo si sliko 3.2.

Algoritem 1 Psevdokoda algoritma Minimax

```
1: minimax(level, player)
2: if gameover or level == 0 then
3:     return score
4: end if
5: children ←all legal moves for this player
6: if player is computer then
7:     bestScore ←-inf
8:     for each child do
9:         score ←minimax(level - 1, opponent)
10:        if score >bestScore then
11:            bestScore ←score
12:        end if
13:    end for
14:    return bestScore
15: else if player is opponent then
16:    bestScore ←+inf
17:    for each child do
18:        score ←minimax(level - 1, computer)
19:        if score <bestScore then
20:            bestScore ←score
21:        end if
22:    end for
23:    return bestScore
24: end if
25: minimax(2, computer) {initial call}
```



Slika 3.1: Slika prikazuje delovanje algoritma Minimax. Modra vozlišča (krogci) predstavljajo poteze nasprotnika, zelena vozlišča (kvadratki) pa poteze pametnega igralca. Na zadnjem nivoju (listi) se tako na starša propagira vrednost otroka z minimalno vrednostjo. Na predzadnjem nivoju se na starša prenese maksimalna vrednost.



Slika 3.2: Simulacija algoritma Minimax z rezanjem alfa-beta.

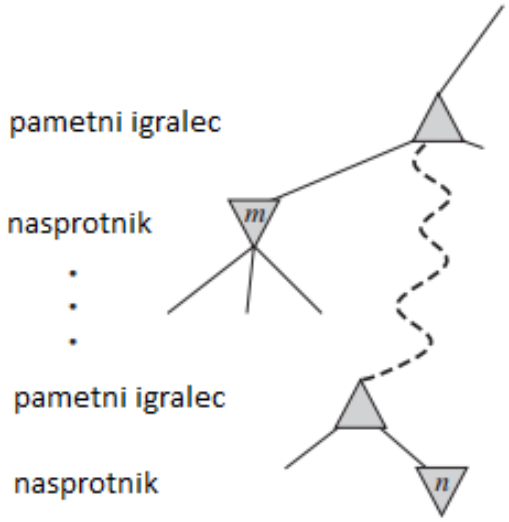
Neraziskani vozlišči na sliki poimenujmo z x in y. Vrednost korenskega

vozlišča znaša:

$$\begin{aligned} \text{Minimax(root)} &= \max(\min(3,12,8), \min(2,x,y), \min(14,5,2)) \\ &= \max(3, \min(2,x,y), 2) \\ &= \max(3,z,2) \\ &= 3 \end{aligned} \tag{3.1}$$

Vidimo, da je vrednost z v enakosti (3.1) manjša ali enaka 2, kar pa pomeni, da nas vrednosti x in y pravzaprav ne zanimata in je končna vrednost algoritma Minimax neodvisna od njune vrednosti.

Poskusimo razložiti še na bolj naraven način: imamo vozlišče n na poljubni globini preiskovalnega drevesa, kamor se lahko premakne igralec. Če ima igralec boljšo možnost m na staršu vozlišča n ali pa kjerkoli višje v drevesu; potem n nikoli ne bo obiskano v realni igri (slika 3.3).



Slika 3.3: Simulacija algoritma alfa-beta. Če je m boljše od n za trenutnega igralca, potem v igri nikoli ne bomo obiskali n.

Alfa-beta preiskovanje med preiskovanjem posodablja vrednosti α in β in tako poskrbi za rezanje preostalih vej na trenutnem vozlišču, če je vrednost trenutnega vozlišča slabša od trenutne α ali β za MAX ali MIN.

α se tako nanaša na najboljšo (največjo) vrednost, najdeno kjerkoli v preiskovalnem drevesu na poti za MAX.

β se nanaša na najboljšo (najmanjšo) vrednost, najdeno kjerkoli v preiskovalnem drevesu na poti za MIN.

Psevdokoda 2 prikazuje algoritem Minimax, nadgrajen z rezanjem alfa-beta, povzet po [24].

3.3 Uporaba algoritma Minimax pri igri Yinsh

Pri apliciranju metode Minimax na igro Yinsh smo si morali odgovoriti na naslednja vprašanja:

- Kakšna bo evalvacijska funkcija za stanje igre? Katere parametre bomo upoštevali pri ocenjevanju igralne plošče?
- Kakšna bo globina preiskovanja?
- Kakšen bo izstopni pogoj iz rekurzije?
- Katere podatke bomo shranjevali v vozlišču drevesa?

Evalvacijsko funkcijo smo izoblikovali postopoma. Začeli smo s popolnoma preprostimi funkcijami, katere smo postopoma nadgrajevali v močnejšo in bolj reprezentativno funkcijo, kar pomeni, da bodo parametri upodabljali realno igro. S kompleksnostjo funkcije smo prilagajali tudi globino preiskovanja in deloma tudi izstopni pogoj iz rekurzije.

3.3.1 Tipi ocenjevalnih funkcij

1. **Število žetono**v

   Čeprav število žetonov naše barve ne pomeni nujno prevlade (saj poveča tveganje, da nasprotnik obrne naše žetone v svojo barvo), pa je kljub temu pomemben dejavnik v igri. Zato je bila naša prva, najpreprostejša rešitev sledeča:

Algoritem 2 Psevdokoda algoritma Alfa-Beta

```
1: minimax(level, player, alpha, beta)
2: if gameover or level == 0 then
3:     return score
4: end if
5: children ←all legal moves for this player
6: if player is computer then
7:     for each child do
8:         score ←minimax(level - 1, opponent, alpha, beta)
9:         if score >alpha then
10:            alpha ←score
11:        end if
12:        if alpha >= beta then
13:            break
14:        end if
15:    end for
16:    return alpha
17: else if player is opponent then
18:    for each child do
19:        score ←minimax(level - 1, computer, alpha, beta)
20:        if score <beta then
21:            beta ←score
22:        end if
23:        if alpha >= beta then
24:            break
25:        end if
26:    end for
27:    return beta
28: end if
29: minimax(2, computer, -inf, +inf) {initial call}
```

Ob doseženi globini **pet** preiskovalnega drevesa zaključimo preiskovanje naslednikov vozlišča (**izstopni pogoj**) ter evalviramo našo igralno ploščo po naslednji enačbi:

$$score(node) = numberOfMarkers(AI) - numberOfMarkers(opponent) \tag{3.2}$$

Preiskovanje zaključimo tudi ob vsaki zbrani vrsti petih žetonov (v poljubni barvi). Takrat vozlišče ocenimo s 100 točkami, če je pet žetonov dosegel pametni igralec. Če je pet žetonov dosegel nasprotnik, vozlišču dodelimo -100 točk.

Kljub preprostosti rešitve smo ugotovili, da ta ni slaba. Dana hevristika je omogočila pametno igranje računalniškega igralca. Pri testiranju smo ugotovili, da je rešitev že zmožna premagati neizkušenega človeškega igralca, ki je sposoben kreirati in braniti svoje vrste in obračati vrste nasprotnika.

Vseeno se zavedamo pomanjkljivosti rešitve: rešitev upošteva le število svojih žetonov, ne preglejuje pa, ali so ti žetoni smiselno oblikovani v vrste. Ne preverja, ali so te vrste ustrezno zaščitene. Gre tudi za požrešno hevristiko: igralec lahko pridobi svojo vrsto, a s tem omogoči zmago nasprotnika v naslednji potezi. Rešitev torej ne upošteva teže zmage ali poraza; preverja le vmesne pobrane vrste petih žetonov.

2. **Število žetonov v vrsti**

   Drugi način je nekoliko kompleksnejši. Razvijemo preprost algoritem, ki preverja vse linije igralne plošče in oceni posamezne vrste žetonov. Algoritem v posamezni liniji išče posamezne žetone, prav tako pa vrste po dva, tri, štiri ali pet žetonov. Seveda je dolžino vrste potrebno ustrezno utežiti.

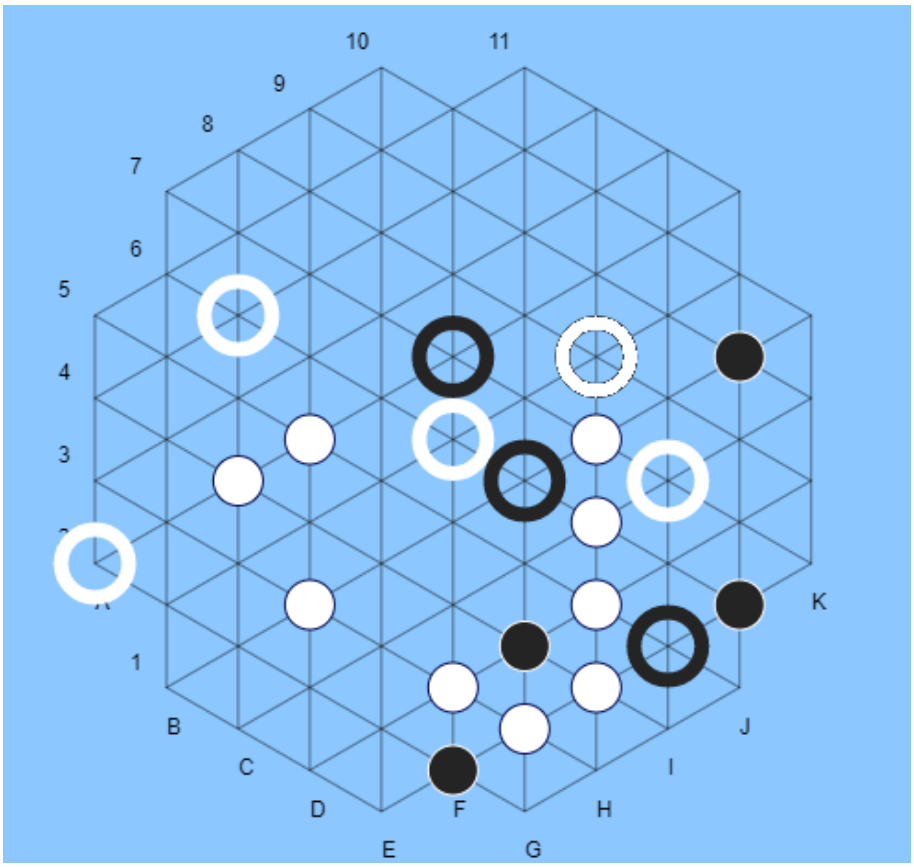
   Poglejmo si sliko 3.4. Vsak naslednji žeton v vrsti prinese večjo vrednost trenutni vrsti. Število žetonov ocenjujemo z vrednostmi 1, 3, 9, 27 in 81 (od 1 do 5 žetonov v vrsti), pri čemer vedno začnemo šteti pri istem žetonu. Če je število žetonov v vrsti n, potem vrsto ocenimo po

enačbi (3.3).

$$score = \frac{1}{2}(3^n - 1) \tag{3.3}$$

En žeton tako prinese eno točko, dva žetona prineseta štiri točke, trije žetoni prinesejo 13 točk, štirje 40 točk in pet žetonov 121 točk. Pametni igralec igra v belih barvah. Najprej si poglejmo bele žetone v navpičnih linijah. Najdemo petkrat po en žeton (pet točk), ter enkrat po štiri žetone v vrsti, kar znaša 40 točk, skupno torej 45 točk.
Podobno naredimo še po obeh diagonalah. Dobimo 13 in 11 točk. Skupna vsota točk za belega igralca na plošči je torej 69 točk.
Podobno izračunamo točke za nasprotnika in jih odštejemo od naših točk. Tako pridemo do končne vrednosti plošče, 57 točk. Na tak način evalviramo igralno ploščo ob naslednjih dogodkih: ko pridemo do globine štiri preiskovalnega drevesa ali pa ob najdbi petih žetonov (ocenimo, preden odstranimo žetone).

Slednji način kaže na že precej izboljšano različico pametnega igralca. Zmožen je premagati že tudi izkušenejše igralce, a vseeno opazimo še nekaj pomanjkljivosti. Tudi tukaj še ne preverjamo zaščite naših žetonov, tj. ali jih nasprotnik lahko v naslednji potezi obrne. Opazimo tudi pojav *zavlačevanja* našega bota. Ker ocenjujemo le stanje celotne plošče in igralca ne nagradimo posebno ob pobranih petih žetonih, pametni igralec spretno priigra zelo močno stanje igralne plošče v svoj prid. Ima veliko kombinacij po štiri žetone, a se včasih zgodi, da ima možnost priti do petih žetonov, pa mu nasprotnik zaradi zavlačevanja to možnost pokvari.



Slika 3.4: Primer igralne plošče.

Kombinacija točk za ocenjevanje vrst žetonov

V okviru opisane funkcije smo naredili manjšo analizo za izbiro kombinacije točk, ki jih podeljujemo vrstam žetonov. Izbirali smo med naslednjimi kombinacijami za 1, 2, 3, 4, 5 žetonov v vrsti:

- 1, 10, 100, 1000, 10000
- 1, 2, 4, 8, 16
- 1, 3, 9, 27, 81
- 1, 2, 3, 4, 5
- 1, 10, 50, 100, 500

Med vsakim parom kombinacij smo izvedli 20 avtomatiziranih iger, pri čemer je vsak pametni igralec igral s svojo ocenjevalno lestvico. Izmed teh dvajsetih iger jih je 10 začel igralec A, drugih 10 iger je začel igralec B (menjava barv igralcev). Igre so imele naključno postavljeno začetno postavitev obročkov. Skupno smo izvedli torej 200 iger, pri čemer je vsak tip kombinacije bil udeležen v 80 igrah. Rezultati so zbrani v tabeli 3.1.

Tabela 3.1: Primerjava kombinacij konstant za ocenjevanje vrst žetonov

Kombinacija	Število zmag	Odstotek [%]
1, 10, 100, 1000, 10000	41 / 80	51,25
1, 2, 4, 8, 16	40 / 80	50
1, 3, 9, 27, 81	54 / 80	64,5
1, 2, 3, 4, 5	19 / 80	23,75
1, 10, 50, 100, 500	46 / 80	57,5

Na podlagi zgornjih rezultatov smo se odločili, da pametnega igralca opremimo s kombinacijo točk 1, 3, 9, 27, 81 za 1, 2, 3, 4, 5 žetonov v vrsti. Če je vrsta daljša od petih žetonov, se vsak naslednji žeton oceni

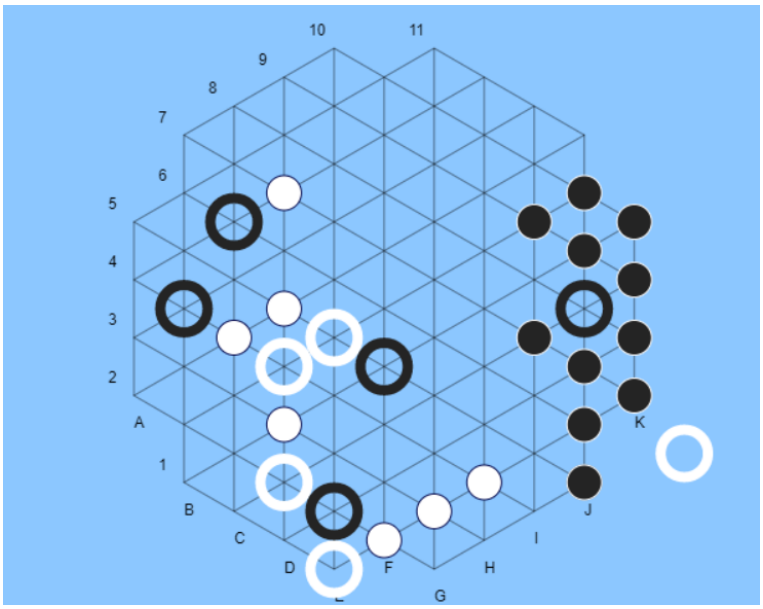
z 81 točkami. Daljših vrst namreč ne smemo oceniti pretirano bolje, saj ne ponazarjajo cilja igre, kar pa je doseganje petih žetonov v vrsti.

3. **Požrešna evalvacijska funkcija**
   Tretja različica je kombinacija prejšnjih dveh. Deluje na naslednji način:

   - Ob doseženi globini štiri preiskovalnega drevesa se vozlišču dodelijo točke po evalvacijski funkciji, opisani v prejšnji točki.
   - Ob pobranih petih žetonih se preiskovanje zaključi in vozlišče oceni z 10000 točkami (pet žetonov pametnega igralca) oziroma -10000 točkami (pet žetonov nasprotnika). Število 10000 tukaj predstavlja največje možno število, s katerim je možno evalvirati igralno ploščo (-10000 je najmanjše). S tem dosežemo, da vedno, ko imamo možnost, poberemo pet žetonov. Prav tako vedno, ko imamo možnost, to preprečimo nasprotniku. V primeru, ko je možno oboje, se odločimo za možnost, ki nam skupno prinaša več točk.

   Igra pametnega igralca s požrešno evalvacijsko funkcijo deluje že bolje - evalvacijska funkcija poskrbi za primerno ocenitev plošče, 10000 točk ob petih žetonih pa poskrbi za to, da igralec vedno pobere svojih pet žetonov in tega ne zavlačuje. Seveda se lahko zgodi, da pobere svojo vrsto, ko bi bilo na primer bolj smiselno preprečiti zmago nasprotniku. Druga pomanjkljivost je ta, da pametni igralec tukaj še ne zna evalvirati zaščite svojih žetonov.

4. **Naprednejša evalvacijska funkcija**
   Poglejmo si sliko 3.5. Če s pomočjo prejšnje evalvacijske funkcije izračunamo število točk, ugotovimo, da v vrsti J najdemo tri zaporedne žetone in še dva zaporedna žetona v črni barvi. A vsak igralec lahko jasno vidi, da obroček, ki ločuje ti dve vrsti, lahko v naslednji potezi doseže vrsto petih (celo šestih) črnih žetonov. Iz te ugotovitve posodobimo našo



Slika 3.5: Primer igralne plošče.

funkcijo: če se v liniji zaporednih žetonov pojavi obroček iste barve, linije preiskovanja ne prekinemo, ampak obroček v vrsti ocenimo s prepolovljeno vrednostjo glede na mesto obročka v zaporedju. V našem primeru je linija dolga kar šest mest. Kot smo že omenili, se šesti žeton oceni enako kot peti. Tako bi linija obročkov med J5 in J10 bila ocenjena s pomočjo vrednosti enačbe (3.4), kar pa že bolj nakazuje na dejansko moč vrste J5-J10.

$$score = 1 + 3 + 9 + 0.5 \cdot 27 + 81 + 81 \tag{3.4}$$

Tudi tukaj vozlišča evalviramo na globini štiri. Poleg posodobitve v sami evalvacijski funkciji, v trenutni različici posodobimo tudi ocenjevanje vrste petih žetonov:
Ko dosežemo linijo petih žetonov, rezultatu evalvacijske funkcije prištejemo še točke po naslednjem kriteriju:

- 100 točk ob dosegu petih žetonov pametnega igralca ali 10000 točk

ob zmagi pametnega igralca. Tukaj želimo ustrezno utežiti zmago v primerjavi s pobrano vrsto petih žetonov.

- -50 točk ob dosegu petih žetonov nasprotnika ali -10000 točk ob zmagi nasprotnika. Tudi tukaj utežimo zmago nasprotnika in le posamezno linijo petih žetonov. Za linijo petih žetonov tukaj dodelimo -50 in ne -100 točk, kot bi lahko pričakovali glede na prejšnjo alinejo.

  S tem parametrom lahko uravnavamo *agresivnost* oziroma *defenzivnost* našega igralca. S trenutno dodelitvijo točk je naša vrsta petih žetonov bolj utežena kot vrsta petih žetonov nasprotnika, kar pomeni, da naš igralec igra nekoliko bolj agresivno. V obratni situaciji bi pametni igralec igral bolj defenzivno. Po krajši empirični analizi se je takšna dodelitev točk izkazala za nekoliko boljšo od dodelitve 100 in -100 točk in dodelitve 50 in -100 točk (defenzivno igranje).

V tej fazi se osredotočimo še na **stabilnost** vrste žetonov. Jasno je, da je vrsta štirih žetonov na liniji K na sliki 3.5 **zaščitena**. To pomeni, da beli nasprotnik v naslednji potezi vrste črnega ne more ogroziti (tj. obrniti žetonov). Včasih se zgodi, da vrsta ni zaščitena in omogoča nasprotniku, da obrne del vrste. Odločimo se nagraditi torej tudi stabilnost vrste:

- 100 točk se prišteje vozliščem, ki pripadajo veji, na kateri točke zaporednih vozlišč naraščajo z globino. Na tak način nagradimo zaporedje potez, ki so v prid našemu igralcu.
- 100 točk se odšteje vozliščem, ki pripadajo veji, na kateri so se med potjo točke dveh zaporednih vozlišč zmanjšale za več kot 100 (kar pomeni, da je nasprotnik obrnil 4 naše žetone).

Z naprednim načinom evalvacije smo prišli že do zelo močne različice našega pametnega igralca, ki rešuje težave, na katere smo naleteli v

prvih treh različicah. Podrobnosti o tem, kako dobra je naša evalvacija, so opisane v poglavju Rezultati.

3.3.2 Globina preiskovalnega drevesa

Globina, ki smo jo dosegali v naštetih tipih pametnega igralca, je bila pet (1. tip igralca) oziroma globina štiri (ostali trije tipi). To so globine, ki smo jih dosegali pred samo optimizacijo algoritma in so odvisne od časa preiskovanja. Glede na to, da smo igre testirali tudi proti človeškemu igralcu, je čas razmišljanja pametnega igralca na potezo moral biti zmeren: v našem primeru je bilo to nekje do 30 sekund, včasih precej manj, včasih tudi več. Po optimizaciji se je ta čas znatno zmanjšal (okoli 5 sekund). Več o tem, kako smo optimizirali čas preiskovanja in s tem povečali globino, je opisano v poglavju Optimizacija.

3.3.3 Struktura vozlišča drevesa

Informacije, ki jih nosi naše vozlišče, so naslednje:

- Indeks igralca (1: beli, 2: črni)
- Stanje igralne plošče (pozicije belih/črnih žetonov/obročev)
- Pozicija, iz katere je trenutni igralec odigral potezo in s tem prišel do stanja igralne plošče
- Pozicija, na katero se je premaknil trenutni igralec in s tem prišel do stanja igralne plošče
- Naslednje trenutno optimalno vozlišče (otrok)
- Stanje stabilnosti vozlišča (da/ne)
- Točke vozlišča (izračunano po evalvacijski funkciji za stanje igralne plošče v trenutnem stanju)

3.4 Drevesno preiskovanje Monte Carlo

Drevesno preiskovanje Monte-Carlo (angl. Monte-Carlo Tree Search) je metoda za iskanje optimalne odločitve v podani domeni. Na podlagi naključnega vzorčenja zbira rezultate ter postopoma zgradi preiskovalno drevo. Sčasoma algoritem postaja boljši pri ocenjevanju vozlišč in posledično najdbi najboljših vozlišč. Metoda se je izkazala kot zelo učinkovito na področju umetne inteligence, predvsem v domeni iger in planiranja [3, 5].

MCTS je sestavljen iz štirih faz (slika 3.6), ki se ponavljajo določeno število iteracij (ponavadi je preiskovanje omejeno s časom) [14]:

1. Izbira (angl. selection)
   V tej fazi iz korenskega vozlišča izberemo enega izmed vozlišč potomcev, pri čemer uporabimo enačbo UCB (3.5). Iz izbranega vozlišča spet izberemo enega izmed potomcev in postopek ponavljamo, dokler ne pridemo do vozlišča, ki še ni v celoti razširjeno (tj. vsi njegovi otroci še niso bili dodani k drevesu) [14].

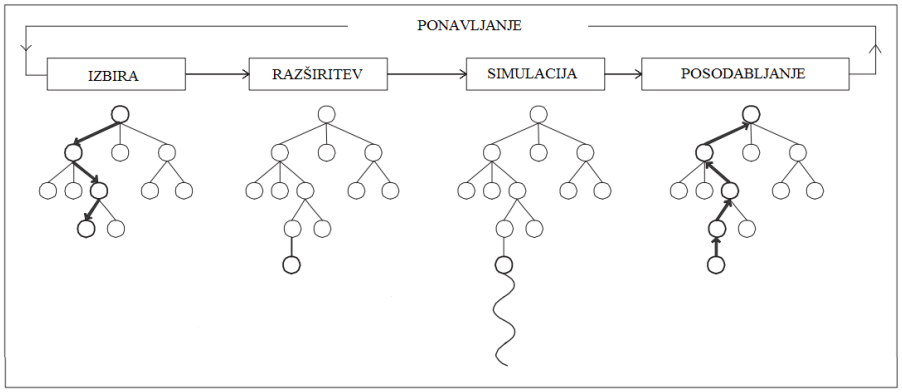
$$v_i = \frac{s_i}{n_i} + C \cdot \sqrt{\frac{\ln(n_p)}{n_i}}, \tag{3.5}$$

   Razložimo še enačbo UCB (angl. Upper Confidence Bounds) (3.5) (vir: [13]). Enačba se uporablja v okviru strategije zgornje meje zaupanja drevesu UCT (angl. Upper Confidence bounds for Trees) [3]. V enačbi s_i določa končno število točk otroka i, kjer je bila zmaga nagrajena z 1 točko, poraz z 0 točkami. Števili n_i in n_p se nanašata na število obiskov otroka i oziroma starša p. Število C je konstanta, ki uravnava razmerje med raziskovanjem (angl. exploration) in izkoriščanjem (angl. exploitation) in se v praksi določi empirično [13].

2. Razširitev (angl. expansion)
   Iz zadnjega izbranega vozlišča prejšnje faze naključno izberemo vozlišče (otroka) L, ki še ni dodano k drevesu [14].

3. Simulacija (angl. playout)
   Iz vozlišča L algoritem simulira igro, ki se začne v stanju vozlišča L, konča pa ob zmagi ali porazu (konec igre). Poteze so lahko naključne, lahko pa bolj odražajo dejansko igranje in tako algoritem doseže bolj zanesljive rezultate, a potrebno je vedeti, da lahko kompleksnejša simulacija zmanjša število simulacij na sekundo in tako v istem času dosežemo manj iteracij MCTS [14]. Potrebno je določiti kompromis med tema dvema parametroma [2].

4. Posodabljanje (angl. backpropagation)
   V tej fazi se propagira rezultat R vse od začetnega lista L do korenskega vozlišča in tako posodobi vsa vozlišča med njima [14]. Posodobi se število obiskov vozlišč in točke vozlišč (a le ob zmagi simulirane igre).



Slika 3.6: Faze algoritma drevesno preiskovanje Monte-Carlo.

3.5 Uporaba algoritma MCTS na igri Yinsh

Za apliciranje metode drevesnega preiskovanja Monte Carlo na igro Yinsh je bilo potrebno nekoliko preoblikovati strukturo vozlišča, ki smo jo uporabljali pri metodi Minimax. Struktura je tako sledeča:

- Indeks igralca (1: beli, 2:črni)

- Stanje igralne plošče (pozicije belih/črnih žetonov/obročev)
- Pozicija, iz katere je trenutno igralec odigral potezo in s tem prišel do stanja igralne plošče
- Pozicija, na katero se je premaknil trenutni igralec in s tem prišel do stanja igralne plošče
- Povezava na starša vozlišča
- Seznam otrok vozlišča
- Število obiskov vozlišča
- Število zmag vozlišča

Čas izvajanja na potezo smo omejili na 10 sekund, kar pomeni različno število iteracij glede na začetno stanje, iz katerega izhaja poteza. Na začetku namreč simulacije trajajo dlje časa, kot bolj proti koncu, ko je rezultat že denimo 2:2. Da omenimo še konkretne številke: število iteracij v 10 sekundah v prvi potezi po postavitvi obročkov znaša v povprečju med 3000 in 4000 iteracij drevesnega preiskovanja Monte-Carlo. V 10. potezi je število iteracij že nad 10000, v zadnjih potezah pa tudi do 500000.

3.5.1 Empirična analiza konstante C

Poglejmo si enačbo (3.5). Prvi del $\frac{s_i}{n_i}$ se osredotoča na izkoriščanje bolje ocenjenih potez, medtem ko se drugi del $\sqrt{\frac{\ln(n_p)}{n_i}}$ posveča raziskovanju novih, še ne raziskanih vozlišč [3].

Izbiro konstante C smo v našem programu določili empirično, pri čemer smo poskusili z naslednjimi vrednostmi:

0.5, 1.0, 1.41, 1.8, 2.0, 2.5.

Izvajali smo avtomatsko igranje iger, pri čemer smo poskušali po 10 iger za vsako kombinacijo konstant. Največ zmag je prinesla vrednost 2.0, zato smo v nadaljevanju vedno uporabili to konstanto.

3.5.2 Izbira tipa simulacije

Kot je bilo že omenjeno, lahko naprednejši tip simulacije povzroči zanesljivejše rezultate. Winands in Nijssen [14] v svoji raziskavi omenjata ϵ-požrešne simulacije (angl. ϵ-greedy playouts), kjer se z verjetnostjo ϵ poteza simulacije naredi naključno. V nasprotnem primeru se uporabi hevristika, ki odraža dejansko igro.

Odločili smo se za preprosto hevristiko, ki je pravzaprav vzeta iz prvega tipa evalvacijske funkcije, omenjene v poglavju 3.3. Igralec, ki je v simulaciji na vrsti, tako z verjetnostjo $1 - \epsilon$ izbere potezo, ki mu v naslednjem koraku prinese največjo število žetonov.

To je hevristika, ki se je razmeroma dobro izkazala že v metodi Minimax in je hkrati časovno povsem nezahtevna.

V avtomatskem testiranju se je različica drevesnega preiskovanja Monte-Carlo z naprednejšemi simulacijami izkazala nekoliko bolje od različice s povsem naključnimi simulacijami (19 zmag od skupno 30 iger). Zato za nadaljnje testiranje uporabimo različico z ϵ-požrešnimi simulacijami.

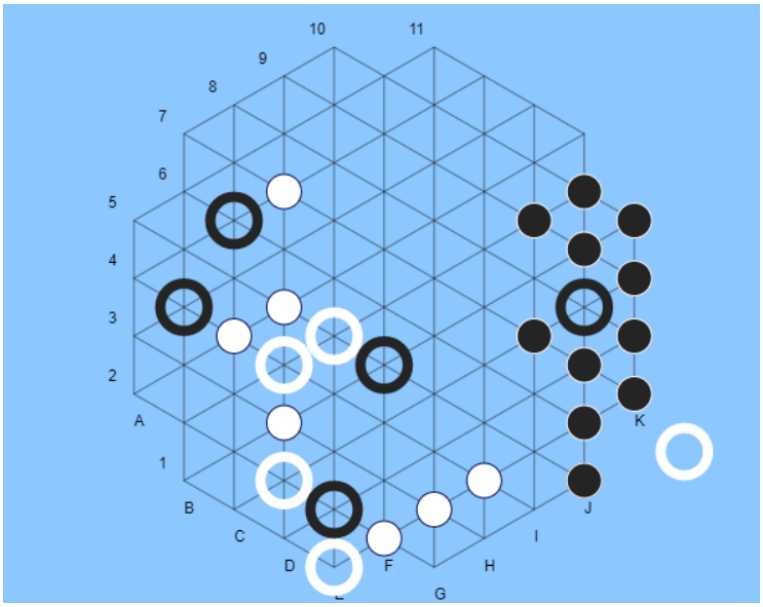
3.6 Začetna postavitev obročkov

Za razliko od klasičnih abstraktnih iger, kot sta šah in dama, ima igra Yinsh spremenljivo začetno postavitev. Pravzaprav je na začetku možnih na milijone različnih postavitev [7], kar pa po našem mnenju pomeni, da je v tej fazi nesmiselno poteze računati po metodah kot sta Minimax in MCTS. Menimo, da k močni začetni postavitvi pripomore predvsem dobra strategija izkušenih igralcev igre Yinsh. Thomas Debray v opisu svojega pametnega igralca navaja, da je pomembno predvsem, da igralec ostane gibljiv in tako obročkov ne postavlja na isto linijo, hkrati pa jih obdrži blizu središča igralne plošče, kar po mnenju profesionalnih igralcev določa centralno dominanco [26].

3.6.1 Postavitev obročkov glede na nasprotnika

Pred testiranjem izvedemo analizo igranja nasprotnikov. Pri tem pridemo do naslednjih ugotovitev:

- Človeški nasprotnik ima proti pametnemu igralcu Yinsh pomembno prednost: človek pogosto prakticira *robno strategijo*, kar pomeni, da si na robu zgradi neuničljivo mrežo, ki se je ne da obrniti (slika 3.7). Medtem ko v splošnem človek ne more razmišljati do globine raziskovanja računalnika, v primeru robne strategije lahko razmišlja globlje od računalnika. Že s postavitvijo obročkov si lahko zagotovi dobre temelje za postavitev takšne mreže. Če oba igralca poznata to strategijo, si lahko s postavitvijo obročkov preprečita, da si en igralec pribori pomembno premoč na robu igralne plošče.



Slika 3.7: Primer situacije, ki je v Yinshu precej pogosta: igralec si na robu zgradi mrežo, ki se je ne da obrniti.

- Če je začetna postavitev obročkov nasprotnika glede na našo postavitev vedno enaka, mora naš pametni igralec zagotoviti rahlo naključnost, saj

s tem nasprotniku onemogoči, da s časom razvije popolno kombinacijo začetnih pozicij obročkov glede na našo postavitev in si tako pridobi prednost, saj naš igralec za postavitev nima močne strategije. To velja predvsem za človeškega igralca, saj njegove postavitve ne moremo analizirati vnaprej, kot lahko to storimo za obstoječe različice Yinsha.

- Implementacija Davida Petra se je že v igri proti človeškemu igralcu izkazala kot zelo dominantna, če človek ne uporablja robne strategije. Ob uporabi robne strategije lahko človek brez večjih težav premaga pametnega igralca Davida Petra. Zato poskusimo to dejstvo uporabiti tudi pri implementaciji našega pametnega igralca.

Zaradi opisanega različnega obnašanja nasprotnikov, ki bodo uporabljeni kot test proti našemu pametnemu igralcu, se odločimo implementirati dve strategiji za postavitev obročkov:

1. V primeru človeškega nasprotnika želimo v postavitev vpeljati naključnost, hkrati pa nasprotniku želimo preprečiti gradnjo robne mreže. Z gradnjo mreže pogosto poskušajo igrati tudi boti na portalu Boardspace, zato se v teh dveh primerih obročke pametnega igralca odločimo postavljati po naslednjih pravilih:

   - Če je možno, postavimo obroček v neposredno bližino zadnje postavljenega obročka nasprotnika. Pozicija ima do šest možnih sosednjih pozicij. Zaradi želje po dinamičnosti pozicijo izberemo naključno. S postavitvijo v bližino nasprotnikovih obročkov mu (vsaj delno) preprečimo igranje močne robne strategije. Z naključnostjo mu preprečimo, da sčasoma razvije popolno kombinacijo začetnih pozicij obročkov in si tako pridobi pomembno začetno prednost.
   - Če to ni možno, postavimo obroček kar se da v sredino igralne plošče. S tem ohranjamo našo gibljivost in centralno dominanco igralne plošče.

2. V primeru igralcev Davida Petra in Thomasa Debraya opazimo, da nam nasprotnik omogoča nemoteno postavitev obročkov na rob. Zato tukaj poskusimo robno strategijo integrirati v našega igralca: če igralec postavi obroček na rob, ga postavimo zraven. Če igralec obročkov ne postavlja na rob (kot se to zgodi pri implementaciji Davida Petra), postavimo 2 obročka na isti rob, ostale pa čim bližje sredini. Obročka na robu s skupnimi močmi lahko zgradita močno mrežo na robu igralne plošče in se pri tem ščitita. Vseeno je dobro imeti nekaj obročkov na sredini, zato ostale obročke postavimo na sredino igralne plošče.

3.7 Odstranitev obročka

Po vsaki pobrani vrsti žetonov je z igralne plošče potrebno odstraniti obroček svoje barve. Pri odstranitvi obročka je potrebno biti zelo pozoren, saj en obroček manj pomeni manjši nadzor nad igralno ploščo. Poleg tega imajo obročki pomembno funkcijo blokiranja nasprotnika in ščitenja naših vrst. Za odstranjevanje obročkov razvijemo preprost algoritem, ki izbere najmanj koristen obroček glede na trenutno stanje igre.

Za vsak obroček naredimo naslednje: z igralne plošče odstranimo obroček R. Na igralni plošči brez obročka R poiščemo najverjetnejšo potezo nasprotnika. To je poteza, ki nasprotniku prinese največ točk (MIN). Tako imamo za vsak odstranjen obroček število točk, ki bi nam jih ta odstranitev prinesla. Izberemo torej obroček, ki nam po odstranitvi in potezi nasprotnika prinese največ točk.

Poglavje 4

Optimizacija

Omenili smo že, da je metoda Minimax časovno zelo kompleksna. V poglavju Optimizacija se zato posvetimo predvsem metodam, ki smo jih uporabili pri izboljšanju časovnih rezultatov našega pametnega igralca. Te metode nam lahko pohitrijo čas razmišljanja, hkrati pa lahko pripomorejo k doseganju večjih globin preiskovalnega drevesa.

Pri opisu optimizacije se bomo osredotočili predvsem na članek *Solving games: Dependence of applicable solving prodecures* [7], ki igro Yinsh opisuje kot eno najkompleksnejših iger v smislu rešljivosti.

Ovrednotili smo omenjeni članek s preizkušanjem omenjenih metod. Hkrati smo predlagali nekaj svojih metod, ki smo jih integrirali v implementacijo pametnega igralca. Omenili bomo tudi metode, ki jih pri Yinshu ni mogoče implementirati zaradi svoje drugačne narave.

4.1 Razsežnost igre Yinsh

Razsežnost preiskovalnega drevesa igre (angl. the size of the game-tree) se ponavadi oceni s potenciranjem povprečnega števila potez na pozicijo in povprečnega števila vseh potez igre. Pri igri Othello razširitveni faktor tako znaša 10, povprečno število potez pa je 58. Ocenjena velikost preiskovalnega drevesa igre Othello je tako 10^{58} [7]. Razširitveni faktor pri šahu znaša 35,

povprečno število potez pa je 70 [11], kar pomeni, da je velikost preiskovalnega drevesa 35^{70}.

Za igro Yinsh smo do približnih vrednosti prišli na naslednji način: za primer smo vzeli 10 avtomatiziranih iger med požrešnim in naprednim pametnim igralcem. Spremljali smo število vseh potez ter število vseh naslednjih možnih pozicij iz trenutnega stanja. Razširitveni faktor na podlagi te empirične analize znaša 47; število skupnih potez znaša 51. Ocenjena razsežnost preiskovalnega drevesa igre Yinsh tako znaša 47^{51}, kar je precej večje od razsežnosti igre Othello in nekoliko manjše od igre šah. Zaradi takšnih razsežnosti seveda preiskovanje do končnega stanja igre ni možno, zato se ponavadi preiskovanje omeji z globino (angl. depth of search; horizon).

4.2 Zbirka otvoritev

Kljub mnogim evalvacijskim in preiskovalnim izboljšavam algoritmov mnogo programov še vedno kaže pomanjkljivosti predvsem pri otvoritvenih fazah same igre, saj programi pogosto nimajo zadovoljivega strateškega načrtovanja. Za reševanje tega problema se pogosto uporabljajo zbirke otvoritev, kjer so shranjena zaporedja potez, ki uveljavijo dobro začetno strategijo [4].

V nasprotju z igrami kot sta šah in dama, ima igra Yinsh dinamično otvoritev. Do tega pride zaradi postavitve obročkov, ki ni znana vnaprej (tako kot je na primer znana začetna postavitev šaha).

Če bi želeli aplicirati metodo zbiranja otvoritev na igro Yinsh, bi morali imeti shranjene začetne poteze za vsako začetno kombinacijo pozicij obročkov. Ker pa je teh kombinacij na milijone, zbirko otvoritev kot optimizacijsko metodo opustimo.

4.3 Transpozicijska tabela

Pri preiskovalnem drevesu igre se lahko pojavi veliko število vozlišč, ki prikazujejo identično stanje igre [7]. Do ponovljivih stanj lahko pride zaradi **transpozicij** – različnih permutacij zaporednih potez, ki pridejo do enakega stanja igralne plošče.

Denimo, da ima Beli na voljo akcijo a_1, na katero lahko Črni odgovori z b_1, neodvisni akciji na drugi strani igralne plošče a_2 pa lahko sledi akcija b_2. Potem zaporedji potez $[a_1, b_1, a_2, b_2]$ in $[a_2, b_2, a_1, b_1]$ končata v istem stanju igralne plošče. Zato si je, če je to le možno, priročno zapomniti, kar je bilo odločenega o tej poziciji, ko je bila nazadnje preiskana, saj si s tem prihranimo ponovno preiskovanje enakega poddrevesa [15] in tako zmanjšamo velikost preiskovalnega prostora. S tem razlogom imajo šahovski pametni igralci ponavadi shranjeno **transpozicijsko tabelo**, ki je implementirana v obliki večje razpršilne tabele (angl. hash table) in hrani informacije o pozicijah, ki so že bile raziskane [25].

Uporaba transpozicijskih tabel ima lahko dramatičen učinek (podvojitev preiskovalne globine pri šahu), potrebno pa je vedeti, da je lahko shranjevanje transpozicijske tabele nepraktično, če evalviramo na milijone vozlišč na sekundo. Na tem mestu se je potrebno odločiti, katera vozlišča shraniti in katera ne [15]. Informacija, shranjena v transpozicijski tabeli za posamezno vozlišče, mora biti kar se da majhna. S tem zagotovimo več shranjenih vozlišč na enako enoto spomina [7].

4.3.1 Apliciranje transpozicijskih tabel na igro Yinsh

V članku [7] je zapisana trditev, da obračanje žetonov naredi transpozicijske tabele neuporabne. To globalno gledano deloma drži, saj zaradi dinamične začetne postavitve in dejstva, da vsaka poteza doda na igralno ploščo nov žeton, res redko pridemo na popolnoma enako stanje igralne plošče.

A potrebno je omeniti vpliv transpozicijskih tabel na preiskovalnem drevesu trenutne poteze, kjer se pogosto zgodi, da na istem nivoju pridemo do

enakega stanja. Poglejmo si sliko 4.1. Denimo, da se beli igralec premakne iz A4 na A5. Sledi premik črnega, ki se premakne iz C6 na C7. Sledi premik belega, ki se premakne iz C4 na C5.
Če beli igralec obrne svoji potezi in se najprej premakne iz C4 na C5, po premiku črnega (C6 na C7) pa se premakne iz A4 na A5, je rezultat identičen. Shema vozlišč preiskovalnega drevesa je prikazana na sliki 4.2

Po tej analizi je tudi jasno, da je takšnih potez, vsaj v takšnem stanju igralne plošče, veliko. Potrebno je le še omeniti, da do istih situacij lahko pride le na istem nivoju preiskovalnega drevesa.

Zato v implementaciji pametnega igralca igre Yinsh uvedemo transpozicijske tabele, ki nastopajo v obliki razpršilnih tabel, pri čemer je ena razpršilna tabela uporabljena za poteze na istem nivoju trenutnega preiskovalnega drevesa. Vozlišč, ki se ponovijo, enostavno ne dodajamo več v preiskovalno drevo.

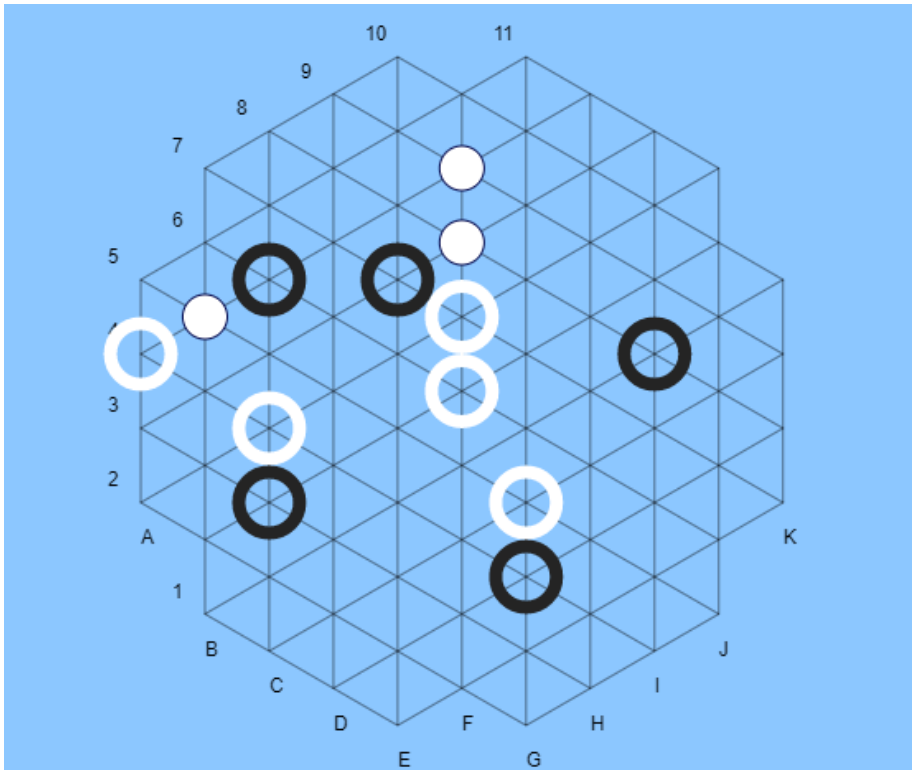
Stanje igralne plošče shranjujemo v obliki niza znakov, ki predstavljajo urejene pozicije vseh obročkov in žetonov po naslednjem pravilu :
obrocki_belega;obrocki_crnega;zetoni_belega;zetoni_crnega.
Za trenutno situacijo bi ta niz bil *A4C4F6F7G5;C3C6E7G4I8;B5F8F9;*.

4.4 Sortiranje potez

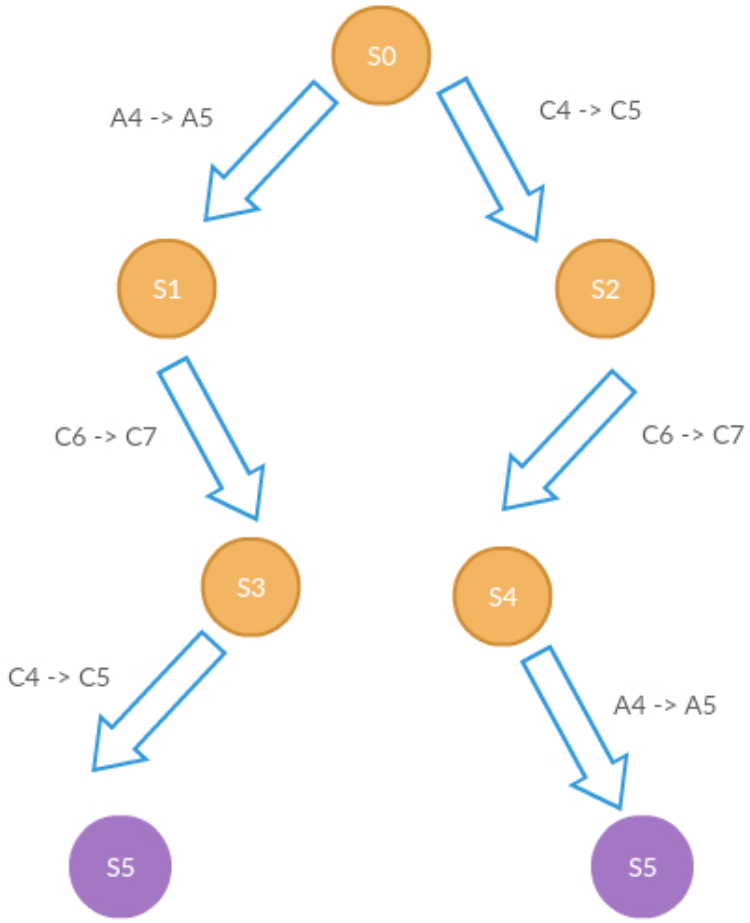
Učinkovitost rezanja alfa-beta je odvisna predvsem od vrstnega reda preiskovanja vozlišč. Poglejmo sliko 4.3. Na drevesu (a) in (b) ne moremo odrezati prav nobenega naslednika vozlišča D, saj je bil najslabši naslednik (iz vidika MIN) generiran najprej. Če bi bil tretji naslednik vozlišča D generiran prvi, bi lahko izpustili preiskovanje drugih dveh vozlišč (rezanje). Sortiranje vozlišč seveda ne more biti izvedeno popolno – če bi to bilo možno, bi sortiranje omogočilo, da pametni igralec igra brez napak [15].

Če bi to bilo možno, bi algoritem alfa-beta lahko raziskal le $O(b^{\frac{m}{2}})$ vozlišč (namesto $O(b^m)$, kar je zahtevnost osnovnega Minimaxa).

To pomeni, da bi se razvejitveni faktor b zmanjšal na $\sqrt{b}$ – v šahu to po-



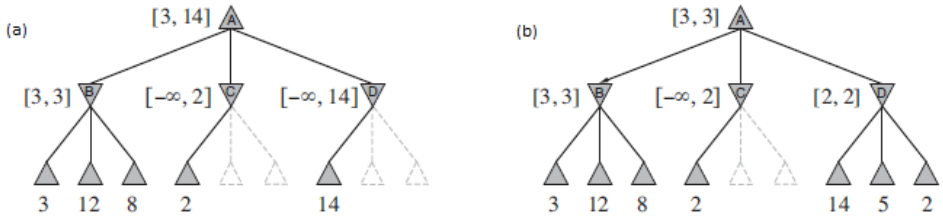
Slika 4.1: Primer stanja igralne plošče.



Slika 4.2: Shema preprostega preiskovalnega drevesa, ki predstavlja transpozicijo. Začetno vozlišče predstavlja stanje igre, prikazano na sliki 4.1. Po obeh prikazanih poteh pridemo do enakega stanja igralne plošče. Na sliki zaradi preprostosti ne prikažemo vseh možnih vozlišč, ki izhajajo iz trenutnega stanja igralne plošče.

meni, da bi se razvejitveni faktor zmanjšal iz 35 na 6. Če so nasledniki pregledani v naključnem vrstnem redu, je število preiskanih vozlišč reda $O(b^{3m/4})$. Pri šahu je že preprosta funkcija za sortiranje dovolj, da dosežemo preiskanih le okoli $O(b^{\frac{m}{2}})$ vozlišč.

Pri sortiranju si lahko pomagamo tudi s preiskovanjem mest, ki so se že izkazali kot zelo dobri v preteklosti. To lahko pomeni prejšnjo potezo, lahko pa preiskovanje trenutnih vozlišč v preteklosti. Pogost način za doseganje slednjega je z *iterativnim poglabljanjem*, kar pomeni, da se v globino pomikamo postopoma – globino povečamo šele, ko smo raziskali že vsa vozlišča trenutnega nivoja drevesa [15].



Slika 4.3: Preiskovalno drevo z rezanjem alfa-beta

Urejanje potez smo integrirali tudi v našega pametnega igralca:
Ob preiskovanju vozlišča najprej ustvarimo vse naslednike, nato pa jim dodelimo točke po enaki evalvacijski funkciji, s katero ocenjujemo končna vozlišča. Vozlišča nato razvrstimo – ob potezi igralca MAX padajoče, ob potezi igralca MIN naraščajoče. Predvidevamo namreč, da bo stanje na nižjem nivoju indikator stanja na nekoliko višjem (globljem) nivoju. Ker najprej raziščemo bolj obetavna vozlišča (iz vidika trenutnega igralca), bomo lahko odrezali več takih vej, ki bodo trenutnemu igralcu zagotovo prinesle manjše število točk in nas zato niti ne zanimajo. To nam omogoča narava algoritma alfa-beta, opisana z enačbo (3.1). Kljub temu, da sortiranje in evalviranje vseh vozlišč prinaša dodatno kompleksnost, ugotovimo, da tehnika znatno zmanjša prostor in čas preiskovanja. Podrobnosti optimizacije bomo opisali v razdelku 4.7.

4.5 Izbira nivoja glede na razvitost igre

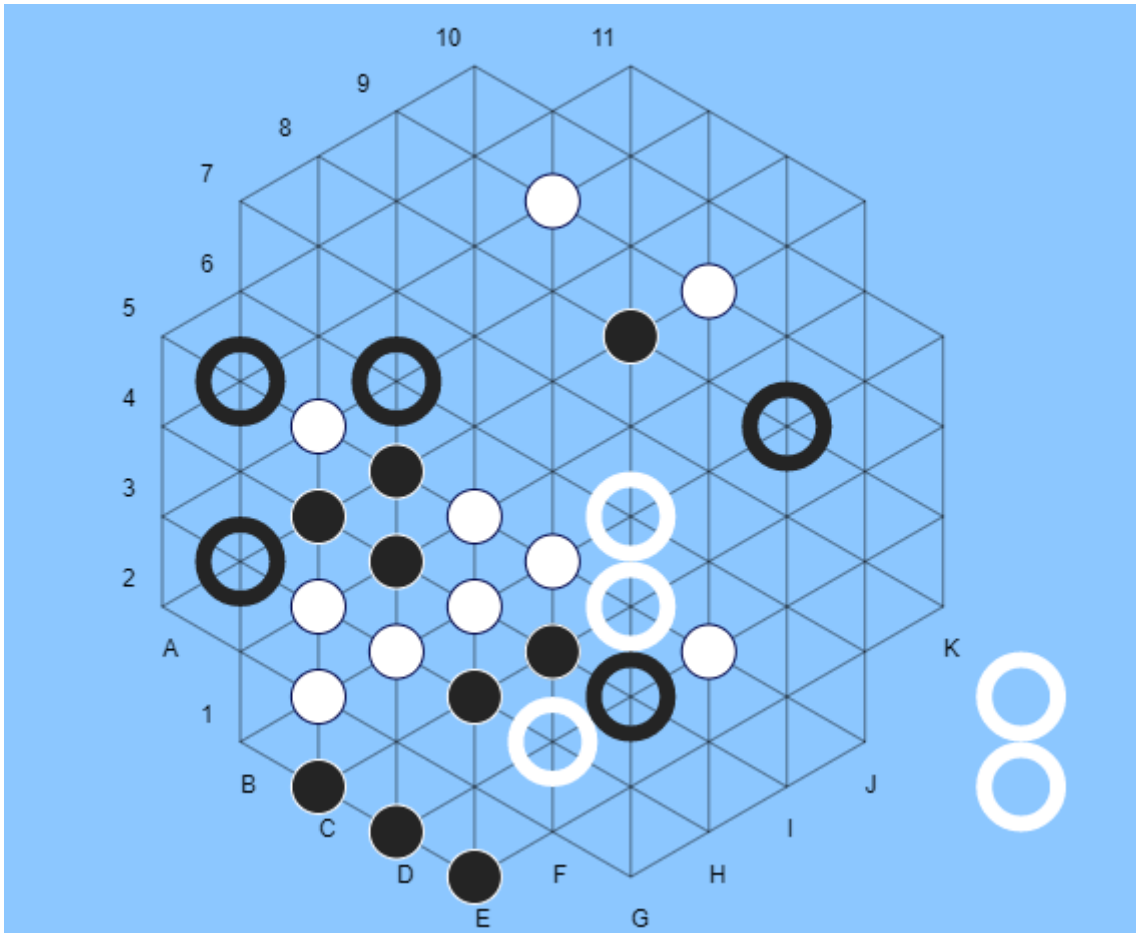
Yinsh je tip igre, kjer se velikost preiskovalnega drevesa manjša s potekom igre (angl. end-games). V igri se namreč zmanjšuje vejitveni faktor drevesa, saj število naslednjih možnih potez tekom igre pada. Tako je denimo povprečno število naslednjih potez pri prvi potezi igre kar 80. Pri končnih potezah se to število zmanjša na 25. To se zgodi zaradi zmanjšanega števila obročkov in zaradi napolnjenosti igralne plošče, kar igralcu omejuje premikanje obročkov.

Odločili smo se, da bomo to informacijo uporabili pri izgradnji našega preiskovalnega drevesa. Brez metod, opisanih v poglavju Optimizacija, smo v zglednem času (pod 10 s) dosegali štiri nivoje preiskovanja. Z vsemi opisanimi optimizacijami ter z uporabo informacije o zmanjševanju preiskovanega drevesa tekom igre naredimo empirično analizo za globino preiskovanja. Pametnemu igralcu določimo naslednje globine:

- Globina 4, če je število potez igre manjše od 15 in če je število naslednikov danega stanja večje od 60.
- Globina 5, če je število naslednikov danega stanja manjše od 60.
- Globina 6, če je število naslednikov danega stanja manjše od 15.

4.6 Grožnje in priložnosti

V igrah pogosto pride do situacije, kjer smo ogroženi z nenadnim porazom ali nenadno zmago (v šahu je to pozicija *šah*, ki pomeni napad na kralja). Takšne situacije lahko znatno zmanjšajo preiskovalni prostor, saj mora nasprotnik posodobiti svoje prioritete in v prvi vrsti preprečiti nasprotnikovo zmago. Tudi v igri Yinsh lahko pride do te situacije. Če na sliki 4.4 črni igralec belemu ne prepreči postavitve petega žetona v vrsti, bo ta v naslednji potezi osvojil tretji obroček in tako zmagal.



Slika 4.4: Primer priložnosti belega igralca, da v naslednji potezi zmaga. Hkrati to pomeni grožnjo črnemu igralcu, da bo poražen.

V članku [7] uporabo strategij za grožnjo in zmago ocenijo kot skoraj neuporabno, saj se zgodijo zelo redko. Yinsh namreč zahteva pobrane tri obročke, prav tako pa igralec z novo vrsto oslabi svoj položaj, saj se mu z enim obročkom manj zmanjša mobilnost, prav tako pa izgubi pet žetonov svoje barve.

Kljub temu, da to drži, se ne strinjamo s trditvijo, da je strategija nenadne zmage ali poraza neučinkovita. V naši napredni evalvacijski funkciji, omenjeni v razdelku 3.3.1, to upoštevamo in tako dodelimo 10000 točk za zmago in -10000 točk za poraz; za pobranih pet žetonov vozlišče ovrednotimo s 100

točkami, ob nasprotnikovi vrsti petih pa -50 točkami. S tem dosežemo boljše ravnotežje med pobrano vrsto žetonov in zmago ali porazom. Tako na primer dosežemo, da ob smo ob doseženem rezultatu 2:1 za nasprotnika bolj osredotočeni na blokiranje zmage nasprotnika, doseganje naše vrste pa je tukaj sekundarnega pomena. Če lahko nasprotnik v naslednji potezi zmaga, mu to poskusimo preprečiti. Če nismo v nevarnosti, pa se spet osredotočimo bolj na pridobivanje vrst našega igralca.

4.7 Vpliv optimizacijskih metod na preiskovanje

V tem poglavju nas bo zanimalo predvsem, kako so zgoraj naštete optimizacije vplivale na pohitritev delovanja našega pametnega igralca. Zanimalo nas bo, kako na določeno stanje igre vpliva integracija sortiranja potez, transpozicijskih tabel in kombinacija obeh optimizacijskih metod. Navedemo tudi, kakšno je stanje brez vseh optimizacijskih metod. Zanimal nas bo čas izvajanja, nivo preiskovanja, število preiskanih vozlišč in število klicev evalvacijske funkcije.

Zadnji parameter smo dodali iz naslednjega razloga: če je preiskano vozlišče *list* drevesa, izračunamo le oceno trenutnega stanja igralne plošče in to vrnemo. Če preiskano vozlišče ni list drevesa, se zaradi potreb sortiranja in ocenjevanja stabilnosti izračunajo ocene vseh otrok vozlišča po enaki formuli, kot se oceni stanje v listih. Tako je časovna zahtevnost preiskovanja listov precej manjša kot časovna zahtevnost preiskovanja staršev, ki morajo evalvirati vse svoje otroke. Število preiskanih vozlišč tako ni zadostna informacija za razlago trajanja preiskovanja, zato smo v tabelah podali še število klicev evalvacijske funkcije.

Oglejmo si slike 4.5, 4.6 in 4.7. Črni igralec ima v teh treh stanjih možnih 88 (slika 4.5), 68 (slika 4.6) in 20 (slika 4.7) potez. Nivo preiskovanja v testiranju teh treh stanj znaša štiri, v zadnjem primeru pa smo zaradi transparentnosti dodali še meritve z globino pet. Tabele 4.1, 4.2, 4.3 in 4.4 povzemajo,

koliko časa je potreboval posamezen pristop za določanje naslednje najboljše poteze.

Z optimizacijami v vsakem primeru dosežemo znatno zmanjšanje števila raziskanih vozlišč, kar pa pomeni tudi znatno pohitritev časa izvajanja. V primeru situacije iz slike 4.7 bi v dejanski igri lahko raziskali tudi do globine šest, saj bi čas izvajanja znašal okoli 4 sekunde.

Iz tabel je jasno razvidno tudi manjšanje preiskovalnega drevesa tekom igre: vidimo, da je v začetnih fazah igre število preiskanih vozlišč drevesa globine štiri brez optimizacij nad 300000. Število raziskanih vozlišč drevesa globine štiri v vmesnih fazah znaša okoli 40000. Število raziskanih vozlišč drevesa globine štiri v končnih fazah znaša okoli 2000.

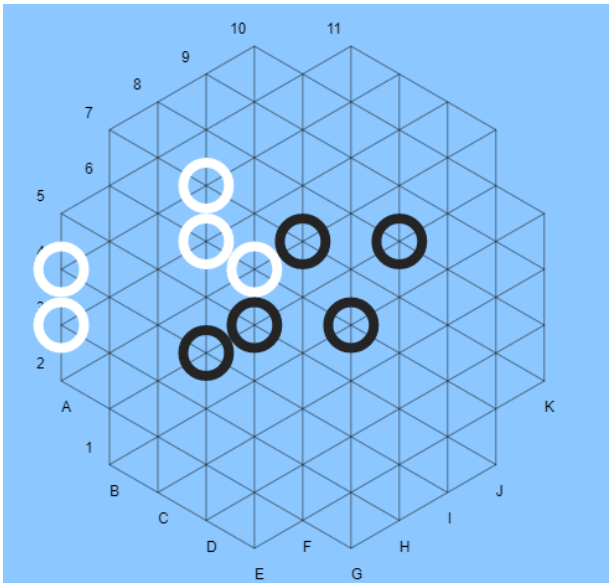
Kot najboljše sredstvo optimizacije ocenimo **sortiranje potez** s pomočjo napredne evalvacijske funkcije. Kljub večji kompleksnosti sortiranja je iz tabel jasno razvidno, da sortiranje potez omogoči znatno zmanjšanje preiskovalnega drevesa. Vozlišča na prvem nivoju so tako lahko v veliki meri indikator vozlišč na višjih nivojih.

Tudi transpozicijske tabele so zmanjšale velikost preiskovalnega drevesa, a v manjši meri. Tako se je na sliki 4.7 število raziskanih vozlišč na nivoju pet s pomočjo transpozicijskih tabel zmanjšalo iz 35070 na 23646. Čas se je zmanjšal iz 2876ms le na 2630ms. Transpozicijske tabele so torej pokazale zmanjšanje preiskovalnega drevesa, a čas raziskovanja se zaradi same kompleksnosti tabel ni znatno zmanjšal.

Meritve smo izvajali na računalniku s procesorjem *Intel(R) Core(TM) i5-4200U CPU @ 1.60GHz 2.30 GHz*.

4.7.1 Vpliv optimizacijskih metod na strukturo vozlišča

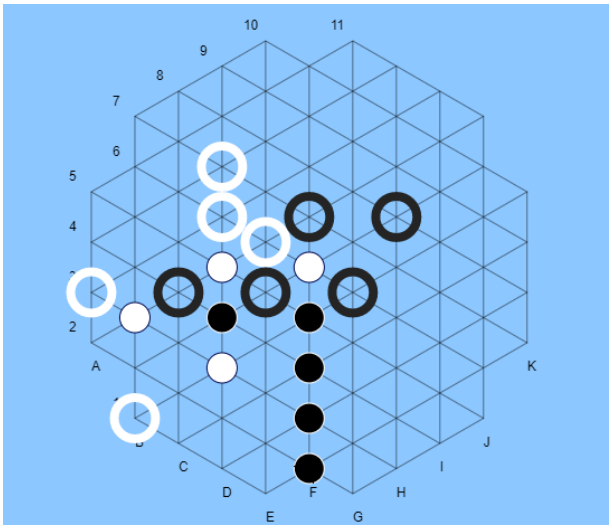
Zaradi apliciranja optimizacijskih metod na našega pametnega igralca, je bilo potrebno deloma spremeniti strukturo našega vozlišča. Tako smo k ostalim lastnostim dodali še seznam vseh naslednikov trenutnega vozlišča. Na takšen način smo lahko zagotovili sortiranje potez.



Slika 4.5: Primer stanja igre, kjer ima črni igralec v naslednji potezi možnih 88 potez.

Tabela 4.1: Analiza optimizacijskih metod za sliko 4.5. Globina preiskovalnega drevesa znaša 4.

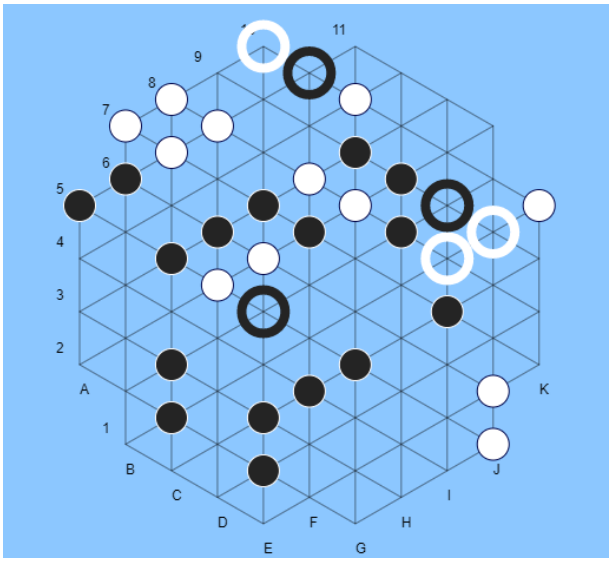
Pristop	Čas izvajanja [ms]	Število raziskanih vozlišč	Število klicev evalvacijske funkcije
Brez optimizacij	24904	308843	1386817
Sortiranje	8038	20613	373332
Transpozicijske tabele	20830	293076	1104506
Kombinacija pristopov	7368	18070	353778



Slika 4.6: Primer stanja igre, kjer ima črni igralec v naslednji potezi možnih 68 potez.

Tabela 4.2: Analiza optimizacijskih metod za sliko 4.6. Globina preiskovalnega drevesa znaša 4.

Pristop	Čas izvajanja [ms]	Število raziskanih vozlišč	Število klicev evalvacijske funkcije
Brez optimizacij	16211	41765	718873
Sortiranje	6863	6836	297442
Transpozicijske tabele	13136	29906	485245
Kombinacija pristopov	5214	12694	254900



Slika 4.7: Primer stanja igre, kjer ima črni igralec v naslednji potezi možnih 20 potez.

Tabela 4.3: Analiza optimizacijskih metod za sliko 4.7. Globina preiskovalnega drevesa znaša 4.

Pristop	Čas izvajanja [ms]	Število raziskanih vozlišč	Število klicev evalvacijske funkcije
Brez optimizacij	853	2194	15507
Sortiranje	699	1064	11778
Transpozicijske tabele	822	1730	11844
Kombinacija pristopov	534	968	10562

Tabela 4.4: Analiza optimizacijskih metod za sliko 4.7. Globina preiskovalnega drevesa znaša 5.

Pristop	Čas izvajanja [ms]	Število raziskanih vozlišč	Število klicev evalvacijske funkcije
Brez optimizacij	2876	35070	97414
Sortiranje	1177	7740	25941
Transpozicijske tabele	2630	23646	63727
Kombinacija pristopov	716	5964	21317

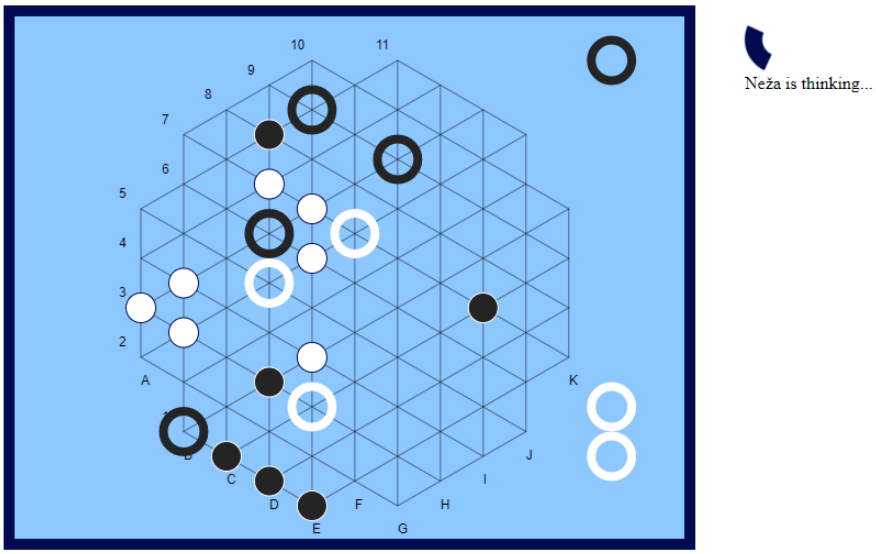
Poglavje 5

Implementacija

Pravila igre ter pametni igralec so bili implementirani v programskem jeziku Java (verzija 8). Za potrebe testiranja proti človeškemu nasprotniku smo implementirali tudi preprost uporabniški vmesnik (slika 5.3), narejen v programskem jeziku Javascript in HTML. Naš zaledni in čelni del komunicirata s pomočjo HTTP zahtevkov, ki se pošiljajo v obliki podatkov v JSON formatu (slika 5.1). Glede na potezo na uporabniškem vmesniku, zaledni del pridobi enega izmed treh oblik ukazov v formatu JSON in odgovori s paketom podobne oblike (slika 5.2). Ob postavitvi obročkov se tako pošlje informacija o igralcu ter o željeni poziciji obročka. Ob premiku se pošlje informacija o igralcu, pozicija obročka, ki se je premaknil in pozicija, na katero se je obroček premaknil. Ob pobiranju petih žetonov se pošlje informacija o igralcu, pozicija obročka, ki se je premaknil, pozicija, na katero se je obroček premaknil ter seznama odstranjenih petih žetonov in obročkov. Tukaj še enkrat poudarimo, da lahko igralec z eno potezo ustvari dve neodvisni vrsti petih žetonov.

Slika 5.1: Slika komunikacije med čelnim in zalednim delom aplikacije.

Slika 5.2: Primeri ukazov v formatu JSON, ki so služili kot komunikacija med zalednim in čelnim delom.



Slika 5.3: Slika uporabniškega vmesnika. Poleg igralne plošče so ob strani prikazani pobrani obročki. Na desni strani je tudi nalagalnik, ki se vrti v času razmišljanja pametnega igralca.

Poglavje 6

Rezultati

6.1 Primerjava napredne, požrešne in MCTS različice pametnega igralca

Pred testiranjem pametnega igralca proti obstoječim nasprotnikom in človeku ugotovimo, katera je naša najmočnejša različica pametnega igralca. Med seboj primerjamo tri različice:

- Minimax s požrešno evalvacijsko funkcijo
- Minimax z napredno evalvacijsko funkcijo
- Drevesno preiskovanje Monte-Carlo z ϵ-požrešnimi simulacijami

Med vsako kombinacijo metod odigramo 30 avtomatskih iger, pri čemer izmenično menjujemo barve igralcev (bela in črna). Rezultati so prikazani v tabeli 6.1. Jasno je, da je Minimax z napredno evalvacijsko funkcijo naša najmočnejša različica, zato se odločimo, da bo ta različica nastopala v testiranju proti računalniškim in človeškim nasprotnikom.

Tabela 6.1: Rezultati primerjave treh različic pametnih igralcev

Igralca		Rezultat
Napredna metoda :	Požrešna metoda	18 : 12
Napredna metoda :	MCTS	28 : 2
Požrešna metoda :	MCTS	28 : 2

Metoda drevesnega preiskovanja Monte-Carlo je bila uspešno aplicirana že na vrsto iger, kot na primer Go, Amazons in Lines of Action [1]. Dobro se je izkazala tudi pri igrah z nepopolno informacijo, kot so na primer Scrabble, Bridge ali Scotland Yard [3]. A še vedno obstaja veliko število tradicionalnih iger, kjer je metoda Minimax primernejša. Taki igri sta na primer šah in dama. MCTS gradi drevo, ki se osredotoča predvsem na najbolj obetavne poti, a kljub temu ni primeren za preiskovalne prostore, ki vsebujejo večje število *pasti* na nižjih nivojih drevesa. Takšne situacije so pogoste v šahu in zahtevajo natančno in taktično igro, da se izognejo porazu. MCTS zaradi svoje naključnosti in načina ocenjevanja lahko zlahka spregleda ali podceni pomembno potezo. Po drugi strani se lahko MCTS izkaže v domenah kot je na primer Go, kjer so pasti redke oziroma se ne pojavijo vse do zadnjih faz igre [1].

Yinsh je simetrična igra s popolno informacijo za dva igralca in je tipa zmaga-poraz (angl. zero-sum game). Tudi Yinsh zaradi obračanja žetonov vsebuje veliko pasti. Igra je po teh lastnostih podobna šahu, zato lahko zgornja obrazložitev zadovoljivo pojasni večjo uspešnost algoritma Minimax proti algoritmu MCTS.

6.2 Portal Boardspace

Portal Boardspace je priljubljen spletni portal za igranje namiznih iger. Možno je igrati proti drugim človeškim nasprotnikom, dodanih pa je tudi nekaj pa-

metnih igralcev iger. Poleg znanih iger kot so šah, dama in Go so na portalu tudi vse igre projekta Gipf.

Na portalu so dodani trije pametni igralci igre Yinsh: *dumbot*, *weakbot* in *smartbot*. Njihov avtor je David Dyer, igralci pa so implementirani s pomočjo iskanja alfa-beta z sprotnim večanjem globine preiskovanja tekom igre. Opisani so kot zelo dobri proti neizkušenim igralcem, a premagljivi proti izkušenejšim igralcem [20].

Z uporabniškim imenom *missRobot* smo se registrirali na portal in s pomočjo implementiranega pametnega igralca, ki uporablja metodo Minimax z napredno evalvacijsko funkcijo, odigrali 10 iger proti *dumbotu* in *weakbotu* ter 20 iger proti *smartbotu*. Rezultati so naslednji:

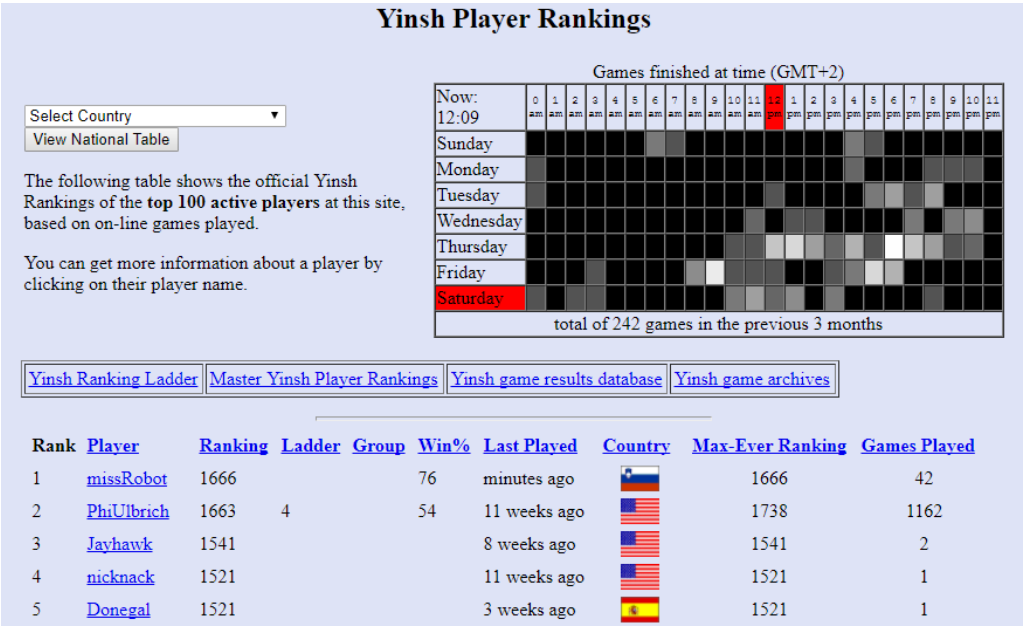
Tabela 6.2: Rezultati pametnega igralca proti botom na portalu Boardspace

Igralca	Rezultat	Odstotek zmag
missRobot : dumbot	8 : 2	80 %
missRobot : weakbot	9 : 1	90 %
missRobot : smartbot	15 : 5	75 %

Po odigranih igrah se tako povzpnemo na trenutni vrh lestvice uspešnosti v igri Yinsh (slika 6.1).

6.3 Implementacija Davida Petra

David Peter je nemški fizik, ki je svojo implementacijo pametnega igralca igre Yinsh objavil na spletu [23]. Svojega pametnega igralca je implementiral z nadgradnjo algoritma alfa-beta, *Negascout*. Negascout zahteva dobro sortiranje potez in tako pohitri delovanje splošnega alfa-beta. Ob slabi sortirni funkciji je algoritem slabši od preiskovanja alfa-beta. Algoritem je prinesel dobre rezultate za računalniške igralce šaha. Prednost algoritma je predvsem



Rank	Player	Ranking	Ladder	Group	Win%	Last Played	Country	Max-Ever Ranking	Games Played
1	missRobot	1666			76	minutes ago		1666	42
2	PhiUlbrich	1663	4		54	11 weeks ago		1738	1162
3	Jayhawk	1541				8 weeks ago		1541	2
4	nicknack	1521				11 weeks ago		1521	1
5	Donegal	1521				3 weeks ago		1521	1

Slika 6.1: Uspešnost našega pametnega igralca na portalu Boardspace.

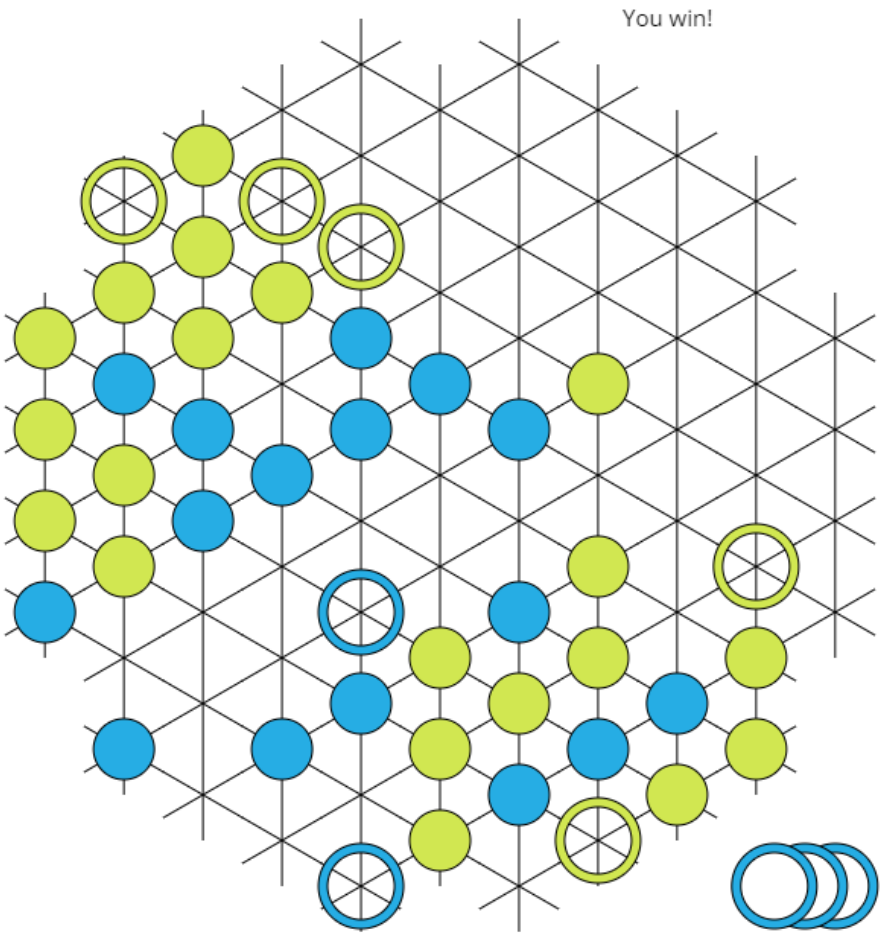
ta, da kljub pohitritvi ne ogrozi natančnosti vrnjenega rezultata in tako vrne enak rezultat, ki bi ga vrnil algoritem Minimax [18].

S pomočjo postavitve obročkov, omenjene v razdelku 3.6.1, ter nagrajevanja stabilnosti vrste, našega igralca spodbudimo, da začne graditi vrste ob robu. Ker se implementacija Davida Petra na kreiranje vrste ob robu odzove prepozno, lahko s to strategijo igralca prepričljivo premagamo.

Implementaciji postavljanja obročkov in samega igranja pri obeh igralcih ne vsebujeta naključnosti, zato je igra enolična in se vedno ponovijo iste poteze. A glede na igranje te igre lahko rečemo, da je naš igralec prepričljivo boljši, saj ga porazi z rezultatom 3:0 (slika 6.2).

6.4 Implementacija Thomasa Debraya

Pametni igralec Thomasa Debraya je leta 2008 sodeloval na Yinsh turnirju in osvojil tretje mesto. Na prvem mestu je bil Pim Nijssen, raziskovalec na maastrichtski univerzi z vrsto pomembnih raziskav na področju umetne in-



Slika 6.2: Zmaga proti implementaciji Davida Petra.

teligence. Tudi T. Debray je v času turnirja pridobival magistrsko izobrazbo na maastrichtski univerzi.

Program, ki je dosegel tretje mesto, je dostopen na njegovi spletni strani [26], kjer so opisane tudi hevristike, ki so bile upoštevane v evalvacijski funkciji:

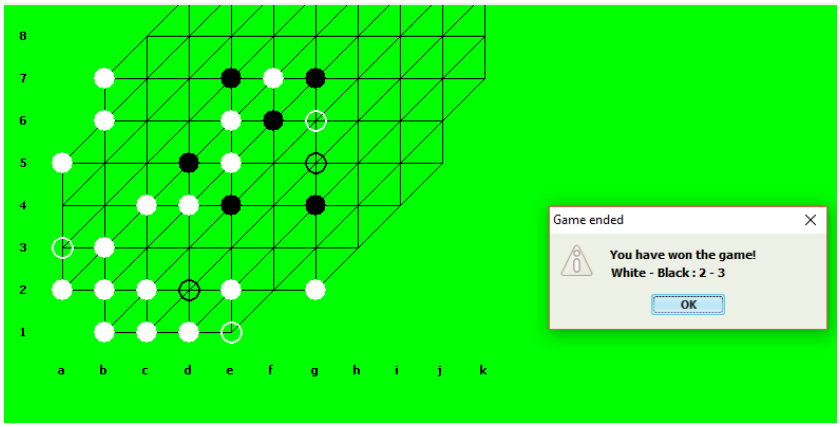
- Število pobranih obročkov.
- Število zaporednih žetonov v vrsti, povezanih z obročkom.
- Razdalja med obročki in središčem igralne plošče. Postavitev obročkov v središču igralne plošče po mnenju Debraya namreč pomeni večjo dominanco in gibljivost na igralni plošči.
- Razdalja med žetoni in središčem igralne plošče.
- Razlika med številom žetonov pametnega igralca in nasprotnika.

Uteži naštetih hevristik so bile optimizirane s pomočjo evolucijskega algoritma z igranjem proti zgodnejšim različicam istega pametnega igralca.

Pametni igralec T. Debraya nam je predstavljal največji izziv. Vseeno opazimo pomanjkljivost, ki smo jo omenili že prej: začetna postavitev obročkov omogoči nemoteno postavitev obročkov na robu. Tudi tukaj je postavitev obročkov na obeh straneh enolična, a v igralcu T.Debraya je vseeno vpletene nekaj naključnosti. Zato smo proti igralcu odigrali 10 iger in jih od tega zmagali 5. Vse igre so bile precej uravnotežene in končane z rezultatom 3:2.

6.5 Človeški nasprotnik

Igro smo testirali tudi proti človeškim igralcem različnih starosti in izkušenj. Neizkušeni igralci, ki poznajo pravila, znajo graditi svoje vrste in obračati nasprotnikove, so bili poraženi že v začetnih fazah razvoja programa (prvi tip evalvacijske funkcije v poglavju 3). Drugi tip igralcev so tisti z več izkušnjami



Slika 6.3: Zmaga proti implementaciji Thomasa Debraya.

in strategijami, ki pa se še vedno ne uvrščajo med najboljše igralce abstraktnih iger. Tudi te ljudi je program brez težav premagal. Program se pokaže kot konkurenčen, a včasih tudi premagljiv nasprotnik najizkušenejšim igralcem Yinsha.

Poglavje 7

Sklep in nadaljnje delo

V pričujočem delu smo analizirali igro Yinsh in njene karakteristike. Implementirali smo več različnih metod za igranje pametnega igralca igre Yinsh. Postopoma smo nadgradili evalvacijsko funkcijo, pri čemer smo upoštevali predvsem število pobranih obročkov in število zaporednih žetonov na igralni plošči. Prišli smo do močne evalvacijske funkcije, ki skupaj s hitrostnimi optimizacijami Minimaxa prinaša več kot zadovoljive rezultate proti obstoječim implementacijam pametnega igralca. Igralec se dobro izkaže tudi proti človeškemu nasprotniku, kjer so mu kos le najizkušenejši igralci igre Yinsh, pa še to le občasno.

Implementirali smo tudi pametnega igralca z drevesnim preiskovanjem Monte-Carlo, ki pa se ne izkaže kot konkurenčen nasprotnik. Menimo, da je razlog predvsem v naravi igre Yinsh, ki je precej podobna šahu. Igra vsebuje več pasti (obračanje žetonov), hkrati pa gre za igro s popolno informacijo za dva igralca in preprostimi pravili. Za takšne igre je ponavadi Minimax še vedno najmočnejša metoda.

V obnašanju našega igralca in konkurenčnih pametnih igralcih opazimo zanimive karakteristike.
Igralci imajo težave z ravnovesjem med dvema dogodkoma, ki pa sta med seboj povezana:

- Pobiranje vrste ali zavlačevanje. Pobrana vrsta hkrati pomeni manjšo

mobilnost, saj izgubimo enega od obročkov. V nekaterih primerih je zavlačevanje s pobiranjem žetonov dobro, saj ohranimo mobilnost in dominanco na igralni plošči. Odstranitev naše vrste lahko omogoči, da takoj zatem zmaga nasprotnik. V tem primeru je bistveno, da najprej preprečimo zmago nasprotnika. Spet v drugih primerih je bistveno, da se vrsta odstrani takoj, sploh če ta ni zaščitena.

- Pobrana vrsta ali zmaga (pobrane 3 vrste). Igra Yinsh ima zelo netipično naravo: pobrane morejo biti tri vrste žetonov. Pogosto se pojavlja vprašanje, kako ovrednotiti vsakega od teh dogodkov, kar se kaže tudi v obnašanju pametnih igralcev, ki včasih ne znajo presoditi, ali je bolje pobrati vrsto ali preprečiti zmago nasprotnika.

7.1 Nadaljnje delo

Zavedamo se šibkosti začetne postavitve obročkov pametnega igralca. Menimo, da bi bilo tukaj potrebnega še veliko raziskovanja in posvetovanja s profesionalnimi igralci, ki bi nam lahko omogočili vpogled v strategije, ki bi igro naredile še močnejšo.

Podrobneje bi lahko analizirali tudi ovrednotenje pobranih vrst in zmage. Kot smo omenili, pametni igralci težko določijo ravnovesje med tem, ali je v neki situaciji bolje vrsto pobrati ali pa s tem počakati. Za kaj takega bi bilo smiselno slediti implementaciji Thomasa Debraya in uporabiti evolucijske algoritme za optimizacijo parametrov našega pametnega igralca.

Glede na to, da smo omejeni na čas izvajanja ob igranju s človeškim nasprotnikom, bi bilo smiselno uporabiti čas razmišljanja človeškega igralca za nadaljnje raziskovanje programa [6]. Program po svoji potezi v času razmišljanja nasprotnika tako izbere najverjetnejšo potezo nasprotnika. Od te poteze naprej začne graditi preiskovalno drevo. V primeru pravilnega predvidevanja igre nasprotnika si tako podaljša čas razmišljanja. Če poteze ni predvidel pravilno, se drevo razvrednoti, preiskovanje pa začne znova.

Dobro bi bilo preveriti tudi patologijo Minimaxa na igri Yinsh. Včasih

namreč prevelika globina preiskovanja povzroči manj zanesljive rezultate. Teoretične analize namreč omenjajo, da z naraščanjem globine narašča tudi šum, ki ga povzroči hevristična funkcija, s katero evalviramo liste drevesa [10].

Zaradi spremenljive narave igre Yinsh bi bilo dobro uporabiti metodo preiskovanja mirovanja (angl. Quiescence search). Pri preiskovanju mirovanja je poudarek predvsem na *nemirnih* vozliščih. To so vozlišča, za katera obstaja velika verjetnost, da se bo ocenjena vrednost igralne plošče v naslednjem koraku znatno spremenila. Takšna vozlišča algoritem preiskuje globlje kot *mirna* vozlišča. Na tak način se lahko znebimo efekta horizontalnosti (angl. horizon effect). Ta se lahko pojavi ob preiskovanju s fiksno globino. List je lahko namreč ocenjen zelo dobro, a v naslednjem koraku se lahko ta vrednost znatno spremeni. Algoritem tako bolj nemirne pozicije preiskuje globlje od mirnih pozicij [17].

Literatura

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

[2] Neža Belej. Analiza preiskovalnih metod na primeru igre scotland yard. Bachelor thesis, Univerza v Ljubljani, Fakulteta za računalništvo in informatiko, 2015.

[3] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[4] Michael Buro. Improving heuristic mini-max search by supervised learning. *Artificial Intelligence*, 134(1-2):85–99, 2002.

[5] Guillaume Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. *Computers and Games*, 5131:60–71, 2008.

[6] James Gillogly. *Performance analysis of the technology chess program.* PhD thesis, Carnegie-Mellon University, 1978.

[7] Marijn JH Heule and Leon JM Rothkrantz. Solving games: Dependence of applicable solving procedures. *Science of Computer Programming*, 67(1):105–124, 2007.

[8] M Tim Jones. *Artificial Intelligence: A Systems Approach*. Upper Saddle River, New Jersey: Pearson Education, Inc. p. 167, 2015.

[9] Paul Kilgo and Dennis Stevenson. Dvonn and game-playing intelligent agents, 2012.

[10] Mitja Luštrek, Ivan Bratko, and Matjaz Gams. Why minimax works: An alternative explanation. In *Proceedings of IJCAI*, pages 212–217, 2005.

[11] T Anthony Marsland. A review of game-tree pruning. *ICCA journal*, 9(1):3–19, 1986.

[12] Frédéric Jürgen Mauss. *Einsatz des Monte-Carlo-Tree-Search-Algorithmus im 2-Personen-Strategiespiel Dvonn*. Technische Universität Berlin, 2012.

[13] J Pim AM Nijssen and Mark HM Winands. Enhancements for multi-player monte-carlo tree search. In *International Conference on Computers and Games*, pages 238–249. Springer, 2010.

[14] Pim Nijssen and Mark HM Winands. Monte carlo tree search for the hide-and-seek game scotland yard. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(4):282–294, 2012.

[15] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach (3rd edition)*. Jones & Bartlett Learning, 2010.

[16] Tomáš Valla and Pavel Veselỳ. Waltz: a strong tzaar-playing program. In *Workshop on Computer Games*, pages 81–96. Springer, 2013.

[17] Diederik Wentink. *Analysis and Implementation of the game Gipf*. PhD thesis, Universiteit Maastricht, 2001.

[18] Algorithm: Negascout. [https://equilibriumofnothing.wordpress.com/2013/10/15/algorithm-negascout/; obiskano 5. avgust 2017].

[19] Board game geek. [https://boardgamegeek.com/; obiskano 5. avgust 2017].

[20] Board space: A place for board games. [http://www.boardspace.net/english/index.shtml; obiskano 5. avgust 2017].

[21] Dvonner. [http://matztias.de/spiele/dvonner/dvonnere.htm; obiskano 5. avgust 2017].

[22] Holtz. [http://holtz.sourceforge.net/; obiskano 5. avgust 2017].

[23] An html5 version of the board game yinsh written in haskell / haste. [https://github.com/sharkdp/yinsh/; obiskano 5. avgust 2017].

[24] Playing against computer with ai. [http://www.ntu.edu.sg/; obiskano 5. avgust 2017].

[25] Transposition table. [https://chessprogramming.wikispaces.com; obiskano 5. avgust 2017].

[26] Yinsh : heuristics. [http://www.netstorm.be/content/2/7/; obiskano 5. avgust 2017].

[27] Yinsh rules. [http://www.boardspace.net/yinsh/english/rules.htm; obiskano 5. avgust 2017].