OTHELLO BOT

Predmet: Računarska inteligencija

Strahinja Sekulić SV34-2020 • Nada Zarić SV63-2020

Sadržaj

Opis implementiranih algoritama

MiniMax algoritam sa Alpha-Betha rezom

MiniMax algoritam sa Alpha-Betha rezom je algoritam za pretragu koji se veoma često koristi u AI igranju igrica da bi se odredio najbolji potez u datom stanju igre.

Algoritam funkcioniše tako što rekurzivno pretražuje stablo igre. Stablo igre predstavlja sva moguća buduća stanja igre sa trenutne pozicije. Stablo se istražuje u dubinu i uz pomoć heurističke funkcije se dodeljuje ocena (procena) svakog čvora stabla. Ova ocena predstavlja koliko je čvor (položaj) povoljan za igrača.

Ovaj algoritam pretpostavlja da igra protiv optimalnog protivnika (pretpostavlja da i protivnik takođe pokušavada napravi najbolji mogući potez za njega) i zbog toga će pokušati da minimizira maksimalni mogući rezultat koji protivnik može da postigne.

Alpha-Betha orezivanje je tehnika koja se koristi da bi se MiniMax algoritam ubrzao tako što se vrši obrezivanje stabla za koje je malo verovatno da će dovesti do optimalnog poteza. Funkcioniše tako što održava dve vrednosti: *alpha* i *betha* (predstavljaju maksimalni/minimalni rezultat koji su do tog trenutka pronađeni na trenutnoj putanji pretrage). Tokom pretrage, ako se utvrdi da je rezultat čvora lošiji od *alpha* vrednosti (tj. to je lošiji potez za trenutnog igrača od drugih poteza koji su do sada istraženi), onda se čvor obrezuje i njegova deca se ne istražuju. Slično tome, ako se utvrdi da je rezultat čvora bolji od trenutne *betha* vrednosti, taj čvor se obrezuje.

Monte Carlo Tree Search

Kao i MiniMax algoritam, Monte Carlo Tree Search (MCTS) algoritam se često koristi za AI igranje igrica, da bi se pronašao najbolji potez. Pogodan je za igre sa visokim faktorom grananja i igre u kojima nemamo potpun set podataka, kao što su Go, Poker i druge strateške igre.

Algoritam funkcioniše tako što gradi stablo pretrage, gde svaki čvor predstavlja moguće stanje igre, a svaka ivica predstavlja mogući prelazak iz tog stanja. Stablo se gradi postepeno dodavanjem novih čvorova kako pretraga napreduje.

U svakom koraku pretrage, algoritam bira čvor za proširenje balansirajući istraživanje i eksploataciju. Drugim rečima, bira čvorove koji nisu posećivani tako često i koji su obećavali u prošlosti. Ovo se često radi korišćenjem formule koja kombinuje stopu pobeda čvora (na osnovu prethodnih simulacija) i broj poseta čvora. Nakon što je čvor izabran, algoritam simulira nasumično odigravanje od tog čvora do kraja igre. Rezultat odigravanja (tj. da li je igrač pobedio, izgubio ili izjednačio) se širi unazad po stablu da bi se ažurirala statistika čvorova i ivica koje su pronađene.

Ovaj proces simulacije koraka se nastavlja sve dok se ne dostigne unapred definisani budžet za pretragu (kao što je vremensko ograničenje ili broj iteracija). Na kraju pretrage, algoritam bira potez koji vodi do najperspektivnijeg čvora u stablu. Ovo se često radi odabirom čvora sa najvećom stopom pobeda.

Expectimax

Kao i prethodna dva algoritma, Expectimax algoritam se često koristi za AI igranje igara, da bi se pronašao najbolji potez. Za razliku od MiniMax algoritma koji pretpostavlja da protivnik uvek igra optimalno, Expectimax algoritam uzima u obzir mogućnost da protivnik neće u svakom trenutku odigrati optimalan potez (odnosno potez koji je najbolji po njega).

Expectimax algoritam funkcioniše slično kao i MiniMax algoritam, odnosno rekurzivno istražuje stablo pretrage u dubinu i uz pomoć heurističke funkcije određuje koliko je čvor (potez) pogodan za igrača. Međutim, za razliku od MiniMax algoritma, Expectimax algoritam za svaki nelisni čvor u stablu računa srednju vrednost njegove dece (srednja vrednost njihovih heurističkih funkcija) i tu vrednost dodeluje posmatranom čvoru. Na kraju se bira potez koji ima najvišu srednju vrednost dece.

ProbCut

ProbCut je algoritam za pretragu koji se obično koristi za AI igranje igrica kako bi se poboljšala efikasnost alpa-betha obrezivanja. Naročito je pogodan za igre sa visokim stepenom grananja, kao što su “Chess” ili “Othello”.

ProbCut algoritam ima za cilj da smanji broj čvorova koje alpha-betha pretraga mora da pretraži korišćenjem probabilističkih ograničenja. Konkretno, algoritam dodeljuje verovatnoću svakom čvoru u stablu igre na osnovu heurističke vrednosti čvora i dubine čvora u stablu. Ako je verovatnoća da se čvor preseče veća od unapred definisanog praga, onda algoritam uklanja taj čvor i njegove potomke.

Prag je određen parametrom zvanim *delta*, koji se izračunava na osnovu varijanse heurističkih vrednosti na svakom nivou dubine u stablu. Ako heuristička vrednost na određenom nivou dubine imaju veću varijansu, onda će *delta* vrednost biti veća, što znači da je manja verovatnoća da će algoritam smanjiti čvorove na tom nivou dubine.

Implementacija

U sledećem tekstu su opisane najbitnije klase (njihovi parametri i metode)

Game

Klasa koja se bavi logikom igre i njenog korisničkog interfejsa.

Polja:

- `turn` - igrač koji je trenutno na potezu
- `board` - tabla na kojoj se igra igra (njeno trenutno stanje)
- `white` - beli agent (ako postoji)
- `black` - crni agent (ako postoji)
- `game_over` - oznaka (flag) da li je igra završena

Metode:

- `def set_player(self, agent)`
  Ova metoda prima parametar `agent`, koji pretstavlja agenta koji igra igru. Postavlja se agent na poziciju crnog igrača. Ova metoda se koristi kada se igra pokreće u modu čovek - agent, kao i u modu agent - agent (da bi se postavio agent crne boje)

- `def set_players(self, agent_one, agent_two)`
  Metoda prima dva parametra: `agent_one` koji postavlja prosleđenog agenta na poziciju belog igrača i `agent_two` koji postavlja prosleđenog agenta na poziciju crnog igrača (pozivanjem metode `set_player`). Koristi se u modu agent - agent.

- `def stop_game(self)`
  Metoda koja se poziva kada želimo da zaustavimo igru prije kraja. Vraća tablu na početno stanje.

- `def __init_gui(self, master)`
  Metoda koja se koristi za inicijalizaciju GUI-ja. Vrši se iscrtavanje celokupnog korisničkog interfejsa (tabla, figure koje predstavljaju igrače, pomoćne labele za praćenje rezultata, i sl.)

- `def __reset_game(self)`
  Metoda koja resetuje igru, odnosno iscrtava se nova tabla sa 4 figure (tabla sa inicijalnom postavkom figura).

- `def __open_settings(self)`
  Metoda u kojoj je definisan settings menu (menu u kome je moguće podesiti parametre igrača, koji agent će da ogra, sa kojom dubinom, brojem simulacija i slično).

- `def __make_a_move(self, row, col, turn)`
  Metoda prima tri parametra: `row` i `col` - služe za definisanje koordinata polja koje igrač želi odigrati i `turn` - igrač (boja) koji želi odigrati potez. Prvo se proverava da li

potez može da se odigra, ako je potez odgovarajući iscrtava se odgovarajući korisnički interfejs (iscrtava se novo stanje na tabli i ako je kraj igre iscrtava se obaveštenje o tome)

- `def __black_plays(self)`
  Pomoćna metoda u slučaju kada u modu čovek - agent. Pravi delay od 0.25 milisekundi da bi moglo lakše da se prati stanje figura na tabli.

- `def __simulation(self)`
  Metoda za izvršavanje simulacije igre u modu agent - agent.

- `def __draw_board(self)`
  Metoda koja se koristi za iscrtavanje korisničkog interfejsa trenutnog stanja table.

- `def __finish_game(self)`
  Metoda za iscrtavanje korisničkog interfejsa u slučaju kraja igre.

- `def __write_move_to_file(row, col)`
  Pomoćna metoda koju smo koristili za upisivanje podataka od važnosti u pomoćni fajl u toku debbuging procesa.

Board

Klasa koja opisuje stanje table u toku igre.

Polja:
- `board` - čuva trenutno stanje table u obliku matrice (dupla lista)
- `last_move` - poslednji odigran potez
- `legal_moves` - svi mogući legalni potezi koje je moguće poduzeti

Metode:
- `def get_board(self)`
  Metoda koja vraća matricu trenutnog stanja teble.

- `def find_all_legal_moves(self, turn)`
  Metoda prima parametar `turn` koji predstavlja igrača za kojeg metoda pronalazi sve legalne poteze u tom trenutku. Povratna vrednost je lista tuple-ova koji reprezentuku koordinate legalnih poteza.

- `def move(self, row, col, turn)`
  Metoda prima tri parametra: `row` i `col` - služe za definisanje koordinata polja koje igrač želi odigrati i `turn` - igrač (boja) koji želi odigrati potez. U ovoj metodi se vrši odigravanje poteza igrača koji je prosleđen. Prva povratna vrednost je oznaka da li je potez izvodljiv ili ne, a druga povratna vrednost je igrač koji je sledeći na potezu

(može se desiti da protivnik nema legalnih poteza pa će isti igrač odigradi dva poteza uzastopno).

- `def get_corners(self, turn)`
  Metoda prima parametar `turn` koji pretstavlja igrača za kojeg se traže uglovi (broj ugova), odnosno traži se broj uglova na kojem se nalaze figure tog igrača. Povratna vrednost je broj takvih uglova.

- `def get_cells_count(self, turn)`
  Slično kao prethodna metoda, samo se ne traži broj zauzetih uglova nego broj zauzetih polja (ćelija).

- `def is_game_over(self)`
  Metoda koja proverava da li je došlo do kraja igre i tu informaciju vraća kao povratnu vrednost.

- `def return_winner(self)`
  Metoda kao povratnu vrednost vraća igrača koji je pobedio partiju (ili None ako je rezultat izjednačen).

- `def print_board(self)`
  Metoda za iscrtavanje matrice stanja table koja je služila za proces debbug-ovanja.

- `def __new_board(rows, cols)`
  Metoda koja ima parametre `rows` i `cols` koji označavaju veličinu table (broj redova i kolona). Služi za iscrtavanje inicijalne table (tabla sa dve bele i dve crne figure).

- `def __has_legal_moves(self, turn)`
  Metoda kao parametar prima `turn`, odnosno igrača za kojeg proverava da li postoji legalan potez. Povratna vrednost je boolean koji označava da li igrač ima neki legalan potez ili nema niti jedan legalan potez

- `def __is_legal_move(self, row, col, turn)`
  Metoda prima parametre `row` i `col` koji označavaju koordinate poteza za koji se proverava da li je legalan i `turn`, odnosno igrač za kojeg se taj potez proverava. Metoda proverava da li je zadati potez legalan za zadatog igrača, i kao povratnu vrednost vraća boolean (legalan/nije legalan potez).

- `def __try_move(self, row, col, turn)`
  Metoda prima parametre `row` i `col` koji označavaju koordinate poteza koji se treba odigrati i `turn`, odnosno igrač koji odigrava potez. Proverava se (“pokušava”) da li igrač može odigradi zadati potez. Povratna vrednost je lista validnih sekvenci.

- `def __prepare_moves(self, turn)`

Metoda proverava da li za prosleđenog igrača (`turn`) postoje legalni potezi. Ako postoje, na tablu postavlja oznake na polja koje igrač može odigrati (`MOVE` polja) i vraća boolean vrednost `True`, a ako ne postoji samo vraća boolean vrednost `False`.

- `def __play_sequence(self, sequences, turn)`
  Metoda koja prolazi kroz niz sekvenci (`sequences` - niz niyova koordinata polja) i za svaku sekvencu poziva metodu `__place_sequence` koja će biti objašnjena u sledećem paragrafu.

- `def __place_sequences(self, sequence, piece)`
  Metoda na svako polje u prosleđenoj sekvenci (`sequence`) postavlja igrača (`turn`).

- `def opposite_turn(turn)`
  Metoda koja za povratnu vrednost vraća igrača koji je protivnik prosleđenom igraču (`turn`).

Agent

Abstraktna klasa koju implementiraju svi Agenti. Polja i metode koje su ovdje objašnjeni neće se objašnjavati u delu objašnjenja samih agenata.

Polja:
- `board` - trenutno stanje table igre
- `turn` - boja igrača koji je agent

Metode:
- `def make_move(self)`
  Metoda koju redefinišu Agenti u zavisnosti od algoritma koji se u svakom koristi. Ova metoda služi da odigravanje poteza Agenta. Povratna vrednost je potez, odnosno njegove koordinate, koji je agent doneo odluku da će odigrati.

utilis.py

U ovom fajlu se čuva heuristička funkcija. Pošto se heuristička funkcija koristi u više od jednog Agenta, bilo je potrebno eksternaliyovati je, da bismo lakše mogli da je menjamo i na taj način poboljšamo.

RandomAgent

Klasa implementira Agent abstraktnu klasu i redefiniše metodu `make_move`. Ponašanje ovog agenta je sledeće: od svih mogućih poteza se bira random potez.

MinimaxAgent

Klasa implementira Agent abstraktnu klasu i redefiniše metodu `make_move`. Algoritam koji se koristi pri odluci o potezu koji će biti odigran je Minimax algoritam.

Polja:

- `depth` - dubina do koje će algoritam istraživati u toku jednog njegovog poteza

Metode:

- `def __search(self, board, turn, depth)`
  Ova metoda je sam algoritam koji poziva redefinisana metoda `make_move`. U njoj se nalazi logika Minimax algoritma. Parametri koje prima: `board` - trenutno stanje table igre, `turn` - igrač na potezu, `depth` - dubina do koje se pretražuje. Povratna vrednost su koordinate najboljeg pronađenog poteza.

- `def alpha_beta(self,board,turn,depth,alpha,beta,maximizing_turn)`
  Metoda koja vrši alpha-betha obrezivanje stabla igre. Kao parametre prima `board` - stanje table u trenutnoj dubini pretrage, `turn` - igrač koji je u toj dubini pretrage na potezu, `depth` - dubina do koje je algoritam došao (ako je dubina 0 to znači da se pretraga tu završava), `alpha` - trenutna alfa vrednost (maksimizirajuća vrednost), `betha` - trenutna beta vrednost (minimizirajuća vrednost), `maximizing_turn` - oznaka da li je trenutna dubina pretraga maksimizujuća ili minimizujuća (da li se gleda `alpha` ili `betha` vrednost. Kao povratna vrednost vraća se najbolji rezultat.

CarloAgent

Klasa implementira Agent abstraktnu klasu i redefiniše metodu `make_move`. Algoritam koji se koristi pri odluci o potezu koji će biti odigran je Monte Carlo Tree Search algoritam.

Polja:

- `num_simulation` - broj simulacija koje se odrađuje tokom jednog poteza
- `exploration` - faktor istraživanja novih čvorova

Metode:

- `def __simulate(self, board, turn)`
  Metoda prima dva parametra: `board` - trenutno stanje table i `turn` - igrač koji odigrava trenutnu simulaciju. Ova metoda odrađuje simulacije od jednog čvora (table) dok se igra ne završi. Povratna vrednost je broj koji označava da li je čvor doneo pobedu (vrednost je 1), poraz (vrednost je 0) i izjednačen rezultat (vrednost je 0.5)

Node

Pomoćna klasa za algoritam Monte Carlo Tree Search. Predstavlja čvor koji se istražuje u ovom algoritmu.

Polja:

- `exploration` - faktor istraživanja novih čvorova
- `board` - stanje table
- `turn` - igrač koji je na redu da odigra potez
- `parent` - roditelj čvor
- `children` - lista čvorova koji su deca
- `visits` - koliko puta le čvor posećen
- `wins` - koliko puta je čvor doneo pobedu

Metode:

- `def is_fully_expanded(self)`
  Metoda koja proverava da li su sva deca čvorovi napravljena. Ako je broj dece jednak broju legalnih poteza, onda je čvor potpuno “razgranat”. Vraća se `bool` vrednost.

- `def is_terminal(self)`
  Metoda koja proverava da li čvor ustvari čvor sa kojim se završava igra. Vraća se `bool` vrednost.

- `def expand(self)`
  Metoda koja od legalnih koraka trenutnog čvor pravi nove čvorove, odnosno pravi svoju decu. Proširuje listu `children` sa novonapravljenim čvorovima.

- `def select_child(self)`
  Metoda pronalazi i vraća dete čvora koji je najbolji kandidat za dalju pretragu.

- `def back_propagate(self, result)`
  Metoda koj vraća unazad, prema početnom čvoru broj poseta i broj pobeda. Ova metoda se poziva rekurzivno, tako što svako dete pozove ovu metodu nad svojim roditelj čvorom.

ExpectimaxAgent

Klasa implementira Agent abstraktnu klasu i redefiniše metodu `make_move`. Algoritam koji se koristi pri odluci o potezu koji će biti odigran je Expectimax algoritam.

Polja:

- `depth` - dubina koju agent istražuje tokom jednog poteza (istražuje se do te dubine)

Metode:

- `def __expectimax(self, board, turn, depth, maximizing)`
  Parametri koje metoda prima su: `board` - trenutno stanje table, za koju se pronalazi sledeći otimalan potez, `turn` - igrač za kojeg se traži optimalan potez, `depth` - dubina se smanjuje, kada bude `depth == 0` zaustavlja se pretraga, `maximizing` - flag koji označava da li je na redu maksimizovan ili minimizovan potez. Uz pomoć Expectimax algoritma se traži sledeći potez koji će agent da odigra.

ProbCutAgent

Klasa implementira Agent abstraktnu klasu i redefiniše metodu `make_move`. Algoritam koji se koristi pri odluci o potezu koji će biti odigran je ProbCut algoritam.

Polja:

- `depth` - dubina do koje algoritam istražuje
- `probcut` - probcut (delta) - određuje da li je rezultat (score) u dobrom opsegu, odnosno da li je potez prihvatljiv.

Metode:

- `def alpha_beta(self, board, depth, alpha, beta, turn, probcut)`
  Parametri koje metoda prima: `board` - stanje table igre, `depth` - dubina do koje algoritam pretražuje tokom jednog poteza, `alpha` - trunatna alfa (maksimizujuća) vrednost, `betha` - trenutna beta (minimizujuća) vrednos, `turn` - igrač koji je na potezu, `probcut` - dređuje da li je rezultat (score) u dobrom opsegu, odnosno da li je potez prihvatljiv. Metoda uz pomoć ProbCut algoritma određuje sledeći optimalan potez agenta. Koristi se alpha-betha pruning i `probcut` parametar za određivanje sledećeg optimalnog poteza.

Analiza rezultata

Sve algoritme smo testirali tako što smo istoj osobi dali da igra igru protiv različitih agenata i sa različitim parametrima. Sve partije su odigrane pod istim uslovima - na istom hardverskom uređaju i sa istom veličinom table za igru (8x8 polja), gde je igru uvek započinjala test osoba.

MiniMax Agent

Parametar `depth` predstavlja dubinu do koje algoritam istražuje (ako je `depth` = 3 istražuju se sledeća tri koraka u budućnosti)

depth = 2		depth = 3		depth = 4		depth = 5	
W	**B**	**W**	**B**	**W**	**B**	**W**	**B**
31	**32**	9	**55**	**47**	17	12	**50**
47	17	11	**53**	**35**	29	24	**40**
47	17	8	**56**	18	**46**	4	**59**
55	9	20	**44**	**50**	14	3	**61**
40	24	17	**47**	**45**	19	4	**60**

Tabela 1

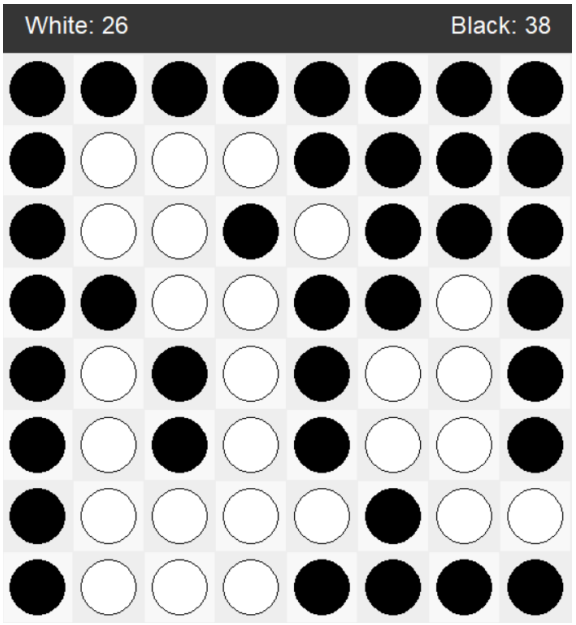
Kao što možemo videti u prethodnoj tabeli (tabela 1), algoritam postiže bolje rezultate kada parametar `depth` ima vrednost koja je neparna. Ovakvo ponašanje smo takođe primetili i kada igraju dva MiniMax algoritma jedan protiv drugog. Ako jedan agent ima `depth` = 3, a drugi agent `depth` = 4 ipak pobeđuje agent koji ima neparan parametar dubine (u ovom primeru `depth` = 3).

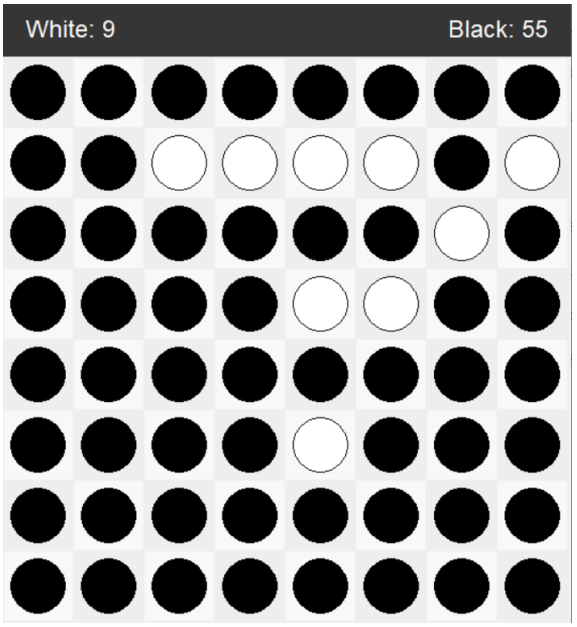
Kada MiniMax algoritam dostigne parnu dubinu, red je da se pomeri protivnik. U ovom trenutku, algoritam pretpostavlja da će protivnik napraviti najbolji potez za sebe, i stoga će izabrati potez koji vodi do minimalnog rezultata trenutnog igrača. To znači da ako trenutni igrač pokušava da maksimizira svoj rezultat, pretraga na parnoj dubini pretpostavlja da će protivnik pokušati da umanji rezultat, što dovodi do konzervativnijeg stila igre.

S druge strane, kada MiniMax algoritam dostigne neparnu dubinu, red je na trenutnog igrača da se pomeri. U ovom trenutku, algoritam pretpostavlja da će trenutni igrač napraviti najbolji potez za sebe i stoga će izabrati potez koji vodi do maksimalnog rezultata za trenutnog igrača. To znači da ako trenutni igrač pokušava da maksimizira svoj rezultat, pretraga po neparnoj dubini pretpostavlja da će trenutni igrač napraviti najagresivniji potez, što će dovesti do agresivnijeg stila igre.

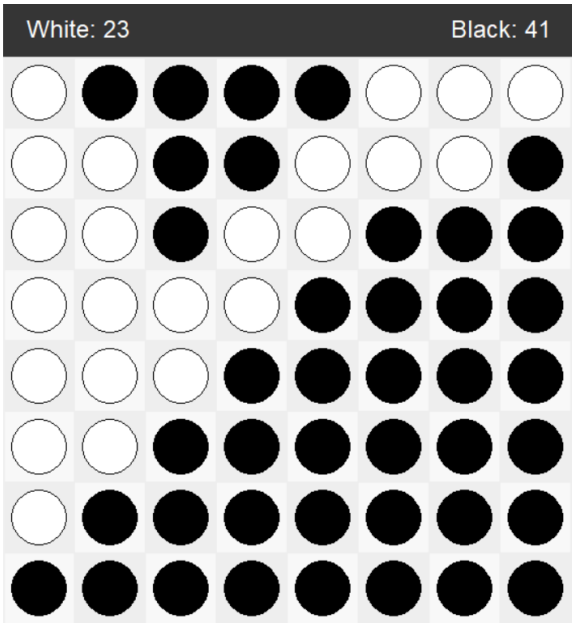
Što se tiče samih preformansi, vreme odigravanja poteza agenta zavisi od dubine do koje agent istražuje stablo (što je dubina veća, potrebno je više vremena da se izračuna sledeći potez agenta. Zauzeće memorije i procesora takođe zavisi od dubine, jer sa povećanjem dubine imamo veći broj koraka koje trebamo istražiti. Kvalitet poteza takođe

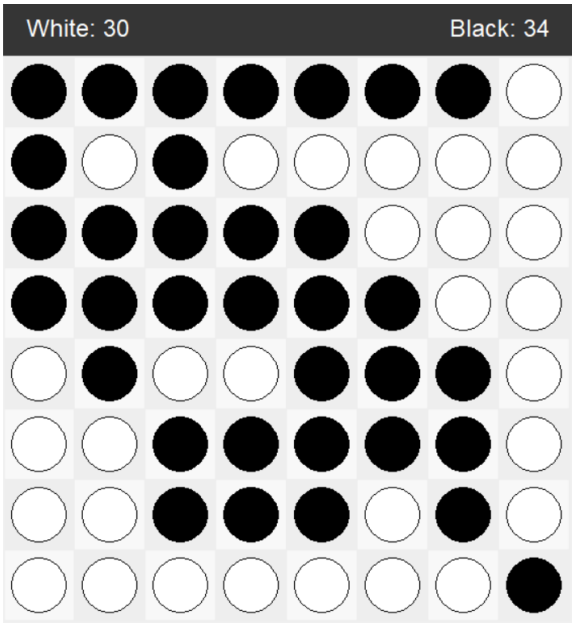
zavisi od dubine, ali i od toga da li je u pitanju parna ili neparna dubina do koje se istražuje (objašnjeni u prethodna dva paragrafa).





Na slikama imamo prikazan primer gde se vidi da Minimax agent sa neparnom dubinom ima bolje ponašanje od Minimax agenta sa parnom dubinom. Na levoj slici, parametar `depth` ima vrednost 4 za belog igrača, dok kod crnog igrača ta vrednost je 3. Na desnoj slici je prikazana situacija kada parametar `depth` ima vrednost 6 za belog igrača i 3 za crnog igrača. Iako u oba slučaja crni igrač ima manju vrednost parametra `depth` on pobeđuje jer ima vrednost parametra `depth` koja je neparna. Ovo ponašanje smo objasnili u prethodnim paragrafima.





Na levoj slici, parametar `depth` ima vrednost 2 za belog igrača, dok kod crnog igrača ta vrednost je 4. Na desnoj slici je prikazana situacija kada parametar `depth` ima vrednost 3

za belog igrača i 5 za crnog igrača. Dakle, u oba slučaja je data “fer” vrednost parametra `depth` (na levoj slici oba parametra su parna, a na desnoj slici oba parametra su neparna). Vidimo da algoritam ima očekivano ponašanje, odnosno da agenti koji istražuju dublje (imaju veću vrednost parametra `depth`) odnose pobedu u igri Othello.

Carlo Agent

Parametar `simulations` (u tabeli označen slovom `s`) predstavlja broj simulacija koje agent treba da izvrši, a parametar `exploration` (u tabeli označen slovom `e`) predstavlja faktor istraživanja (koliko dopuštamo algoritmu da istražuje nove čvorove, a ne samo one koji su namdoneli pozitivne rezultate).

s = 250, e = 0.8		s = 250, e = 1.5		s = 250, e = 2.5	
W	**B**	**W**	**B**	**W**	**B**
15	**49**	**35**	29	**54**	10
35	29	**57**	7	29	**35**
47	17	32	32	31	**33**
42	22	15	**49**	**50**	14
29	**35**	29	**35**	**42**	22

Tabela 2

s = 500, e = 0.8		s = 500, e = 1.5		s = 500, e = 2.5	
W	**B**	**W**	**B**	**W**	**B**
33	31	32	32	10	**54**
33	31	29	**35**	**40**	24
37	27	21	**43**	19	**45**
17	**47**	**56**	8	**48**	16
37	27	16	**48**	**33**	31

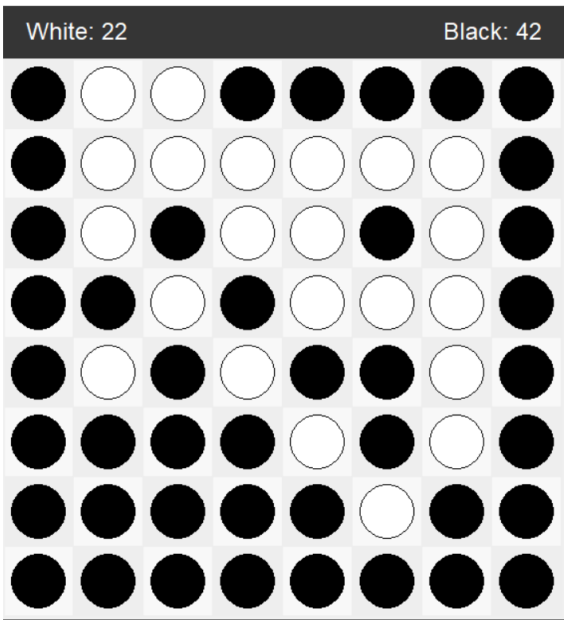
Tabela 3

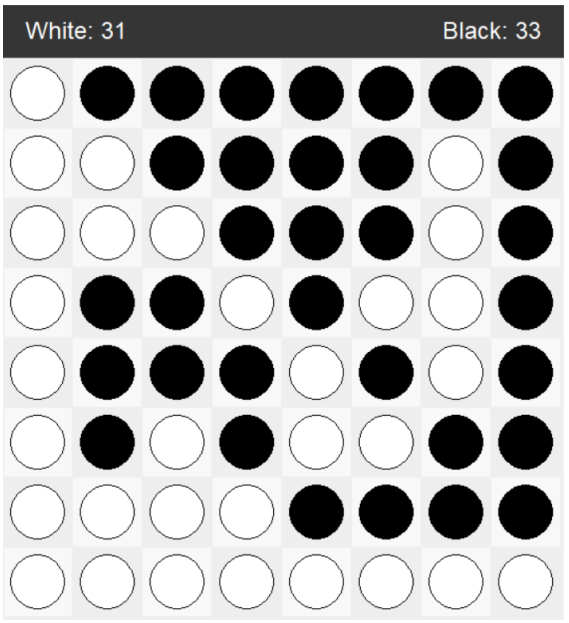
Analizom podataka iz prethodnih tabela (tabela 2 i 3) možemo videti da smo dobili drugačije rezultate nego kod samog MiniMax algoritma. Ovde su rezultati "raznolikiji", odnosno nemamo situaciju da konstantno pobeđuje beli ili crni igrač. Možemo takođe primetiti, da menjanjem parametra `exploration` se menja u uspešnost samog agenta. Ako agentu damo premalo ili previše slobode u istraživanju novih čvorova, to dovodi do gorih rezultata agenta. Zbog toga je potrebno pronaći balans između prevelikog i premalog faktora

istraživanja. Takođe, možemo primetiti da postoji razlika u rezultatima kada imamo različit broj simulacija (iteracija). Sa povećanjem iteracija, rezultati postaju bolji za Carlo Agenta.

Tokom analize rešenja smo pokušavali postaviti različite vrednosti parametra *exploration*, gde smo najbolje rezultate dobijali za `exploration` = 1.5.

Što se tiče preformansi, brzina odigravanja poteza zavisi od broja simulacija (iteracija) koje smo zadali algoritmu. Što je veći broj simulacija to je više vremena potrebno da se odigra potez. Ovo isto važi i za zauzeće memorije i procesora. Parametar `exploration` nema uticaja na same preformanse, nego samo na kvalitet odigranog poteza.





Na prethodne dve slike vrednosti parametara `simulations` i `exploration` su iste (beli igrač: `simulations=50` i `exploration=1.5`, crni igrač: `simulations=50` i `exploration=1.5`). Cilj je bio pokazati da veći broj simulacija dovodi do većeg broja pobeda. Vidimo da u prethodna dva slučaja pobeđuje igrač sa većim brojem simulacija (crni igrač).

Expectimax Agent

Parametar `depth` (u tabeli označen slovom `d`) predstavlja dubinu do koje algoritam istražuje (ako je `depth` = 3 istražuju se sledeća tri koraka u budućnosti).

depth = 2		depth = 3		depth = 4		depth = 5	
W	**B**	**W**	**B**	**W**	**B**	**W**	**B**
53	11	**52**	12	0	**34**	0	**18**
44	20	14	**50**	32	32	22	**42**
31	**33**	**41**	23	29	**35**	15	**49**

46	18	**44**	20	**33**	31	15	**49**
35	29	**43**	21	26	**38**	11	**53**

Tabela 4

Kao što možemo videti u prethodnoj tabeli (tabela 4), Expectimax agent u igri protiv čoveka sa povećanjem vrednosti parametra `depth` povećava se i broj pobeda.

U pogledu preformansi, vreme odigravanja poteza zavisi od dubine do koje agent istražuje potez. Sa povećanjem vrednosti parametra `depth` povećava se i vreme koje je potrbno da agent odigra svoj potez. Ovo se događa zbog toga što sa većom dubinom imamo i veći broj čvorova koje je potrebno istražiti. Takođe, ovaj algoritam nema Alpha-Betha obrezivanje kao MInimax algoritam. Iz tog razloga, kada ga poredimo sa Minimax algoritmom, vreme odigravanja poteza je duže, jer Expectimax agent istražuje više čvorova od Minimax agenta kada imaju istu vrednost parametra `depth`. Što se tiče zauzeća memorije i procesora, zauzeće raste sa povećanjem vrednosti parametra `depth`.

ProbCut Agent

Parametar `depth` (u tabeli označen slovom `d`) predstavlja dubinu do koje algoritam istražuje (ako je `depth` = 3 istražuju se sledeća tri koraka u budućnosti) `probcut` (u tabeli označen sa `p`) određuje da li je rezultat (score) poteza u dobrom opsegu, odnosno da li je potez prihvatljiv.

d = 2, p = 0.1		d = 2, p = 0.9		d = 3, p = 0.1		d = 3, p = 0.9	
W	**B**	**W**	**B**	**W**	**B**	**W**	**B**
51	13	**50**	14	28	**36**	16	**48**
32	32	**35**	29	21	**43**	0	**22**
44	20	**50**	14	18	**46**	**36**	28
24	**40**	**40**	24	14	**50**	18	**46**
20	**44**	30	**34**	18	**31**	11	**53**

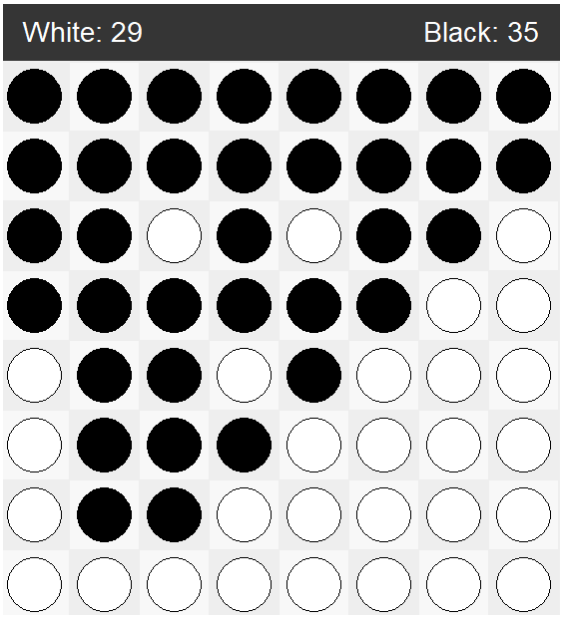
Tabela 5

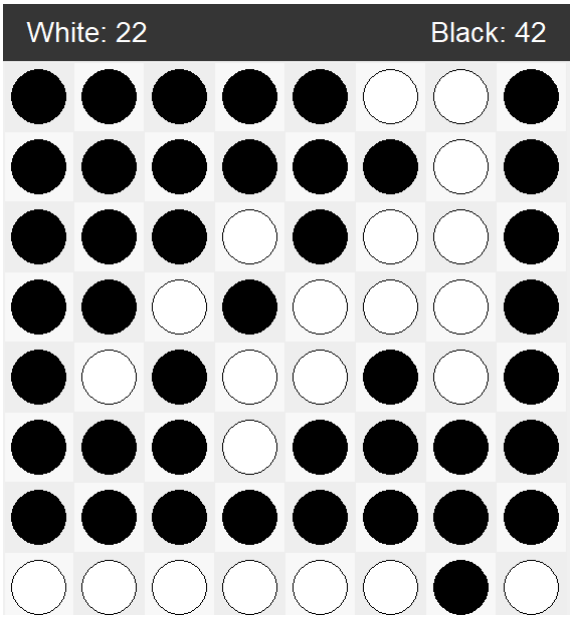
Analiziranjem prethodne tabele rezultata (tabela 5), u poređenju sa tabelom rezultata Minimax algoritma vidimo slično ponašanje kada je dubina do koje se istražuje u pitanju (parametar `depth`). Možemo primetiti, da je čovek odneo veći broj pobeda kada je vrednost ovog parametra paran broj (jer se posmatra minimizirajuća (betha) vrednost), dok manje pobeda ima kada je vrednost parametra neparan broj. S obzirom da je ProbCut algoritam proširenje Minimax algoritma sa Alpha-Betha obrezivanjem, isti zaključak sledi kao što smo napisali u delu analize Minimax algoritma.

Kada je u pitanju parametar `probcut` on ima uticaja na algoritam u smislu koliko ćemo biti rigorozni u pogledu biranja koraka koji dolazi u obzr. Što je veća vrednost ovog parametra, to znači da dopuštamo algoritmu da uzme više koraka u obzir, odnosno više

čvorova će se pronaći koji imaju rezultat čije odstupanje nije veće od zadatog parametra `probcut`. Potrebno je da se pazi da ne postavimo prevelik ovaj parametar, Jer će u tom slučaju algoritam imati slične ili iste preformanse i ponašanje kao i Minimax algoritam.

Što se tiče preformansi, samo vreme, zauzeće memorije i procesora zavise od oba parametra (`depth` i `probcut`). Što je vrednost parametra `depth` veći, to znači da istražujemo više u dubinu. U tom slučaju pretražujemo veći broj čvorova, što znači da nam je potrebno više vremena i drugih resursa za pretragu. Slično ponašanje imamo i sa parametrom `probcut`. Naime, znamo da ProbCut algoritam ustvari poboljšava preformanse samog Minimax algoritma. Uz pomoć ProbCut algoritma, mi zapravo odsecamo čvorove za koje imamo predikciju da je manja verovatnoća da će dovesti do pozitivnog ishoda po igrača. Tako da što nam je `probcut` parametar manji, imaćemo više odsecanja i manje istraživanja čvorova, što dovodi do manjeg vremena odigravanja i manjeg trošenja drugih preformansi. Kako povećavamo ovaj parametar bližimo se ponašanju Minimax algoritma, imamo veći broj čvorova koje istražujemo, što troši više vremena i drugih resursa.





Na prethodne dve slike imamo pregled ponašanja ProbCut algoritma u zavisnosti od vrednosti `probcut` parametra. U oba slučaja parametar `depth` ima vrednost 2. Kod leve slike parametar `probcut` ima iste vrednosti i za belog i za crnog igrača. Možemo primetiti da rezultat približan. Na levoj strani vrednost parametra `probcut` za belog igrača je 0.1 dok je za crnog igrača 0.9. Ovde vidimo da crni igrač ima daleko veći rezultat.