SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 829

Razvoj agenta za igru Othello

Ivan Milinović

Zagreb, srpanj 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 10. ožujka 2023.

ZAVRŠNI ZADATAK br. 829

Pristupnik:	**Ivan Milinović (0036534449)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	doc. dr. sc. Marko Đurasević

Zadatak: **Razvoj agenta za igru Othello**

Opis zadatka:

Proučiti pravila igre Othello te pronaći ili izraditi jednostavnu implementaciju igre koja omogućuje igranje od strane čovjeka ili računalnog agenta. Istražiti postojeću literaturu koja se bavi razvojem agenata za igru Othello. Proučiti mogućnost primjene različitih metoda strojnog učenja koje se mogu iskoristiti za automatski razvoj agenta za igru Othello. Implementirati sustav za automatski razvoj agenata za igru Othello. Ocijeniti učinkovitost razvijenih agenata i predložiti moguća poboljšanja. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Rok za predaju rada: 9. lipnja 2023.

SADRŽAJ

1. Uvod **1**

2. Igra Othello **2**

2.1. Pravila igre . . . 3

2.1.1. Valjani potez . . . 3

2.1.2. Nemogućnost igranja poteza . . . 4

2.1.3. Odabir pobjednika . . . 4

2.1.4. Implementacija . . . 4

3. Neuronska mreža **5**

3.1. Umjetni neuron . . . 5

3.2. Unaprijedna neuronska mreža (eng. *feedforward*) . . . 5

3.3. Linearni model . . . 6

3.3.1. Statička analiza ploče . . . 6

3.3.2. Ocjena slobode igrača . . . 6

3.3.3. Nagrađivanje zauzetih kutova . . . 7

3.4. Drugi model . . . 8

3.4.1. Ulazni sloj . . . 8

3.4.2. Skriveni i izlazni sloj . . . 8

3.5. Minimax algoritam . . . 9

3.5.1. Alfa-beta podrezivanje . . . 9

4. Genetski algoritmi **11**

4.1. Operator križanja . . . 11

4.1.1. Simulirano binarno križanje (SBX) . . . 12

4.2. Operator selekcije . . . 12

4.2.1. Turnirska selekcija . . . 12

4.3. Operator mutacije . . . 12

4.4. Dinamičke mutacije i križanja 13
4.5. Funkcija dobrote 13

5. Implementacija **15**
5.1. Implementacija simulatora 15
5.2. Klasa *Generic_bot* 15
5.3. Klasa *GA_bot* 16
5.4. Klasa *Trainer* 16

6. Rezultati **18**
6.1. Model *GA_NN_bot2* 18
6.2. Model *GA_linear_bot* 19
6.3. Daljnji razvoj 20

7. Zaključak **22**

Literatura **23**

1. Uvod

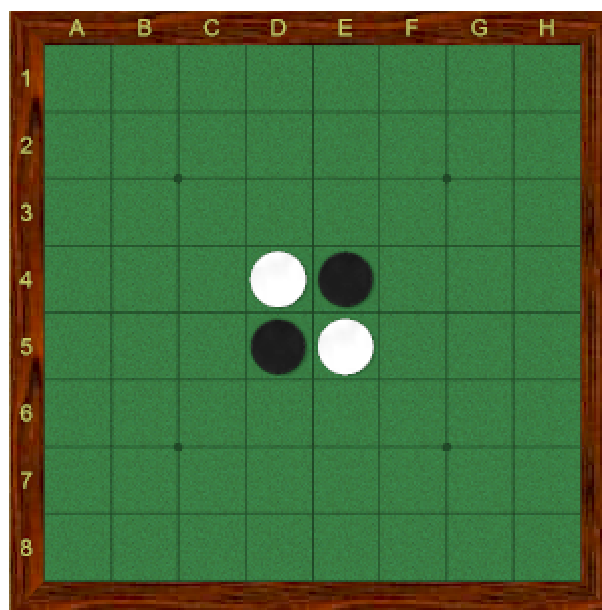
Nedavnom pojavom ChatGPT-a umjetna inteligencija postala je jedna od najpopularnih tema današnjice, no gotovo svi ljudi su već godinama izloženi raznim drugim vrstama umjetne inteligencije. Od ciljanjih reklama i oglasa na internetu do sustava za preporučivanje videozapisa. Umjetnom inteligencijom najčešće se rješavaju problemi koji su presloženi da bi se riješili algoritamski.

Društvene igre već dugo služe kao test ljudske inteligencije, strategije i donošenja odluka. Zahtjevaju od igrača da procjene trenutno stanje igre, predvide protivnikove poteze i prilagode im se. Pojavom umjetne inteligencije društvene igre postale su popularno područje istraživanja njenih mogućnosti.

U ovom radu fokus će biti na primjeni evolucijskog računarstva, konkretnije genetskih algoritama za izradu agenta za igru Othello. U poglavlju 2 objašnjena su pravila igre Othello. U poglavlju 3 nalazi se kratki pregled neuronskih mreža i njihovih modela korištenih u radu. U poglavlju 4 pojašnjeni su genetski algoritmi. U poglavlju 5 prikazan je kratki pregled implementacijske strukture agenta i okoline za njegovo treniranje te u poglavljima 6 i 7 nalaze se rezultati, ideje za budućnost te zaključak.

2. Igra Othello

Othello je igra za 2 igrača koja se igra na ploči koja se sastoji od 64 polja raspoređenih u 8 redova i 8 stupaca. Patentirana je u Japanu 1971. godine, no ona je zapravo moderna varijanta puno starije igre Reversi. Igra se igra sa 64 žetona koji su s jedne strane bijeli, a s druge strane crni. Svaki igrač započinje igru s 32 žetona. Bijeli igrač žetone može postaviti samo na bijelu stranu okrenutu prema gore, a crni na crnu. Na početku partije bijeli i crni igrač postavljaju 2 žetona na ploču kao što je prikazano na slici 2.1. Uvijek je crni prvi na potezu.

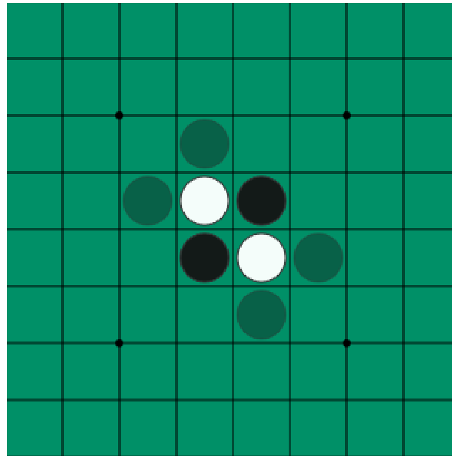


Slika 2.1: Početna konfiguracija ploče

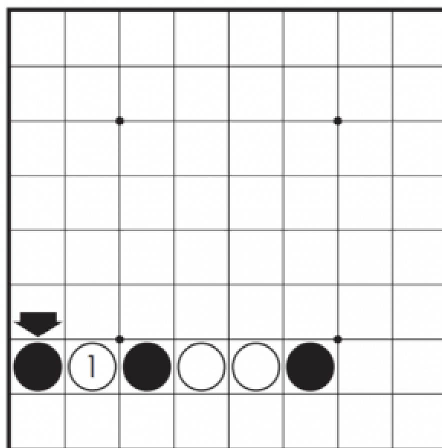
2.1. Pravila igre

2.1.1. Valjani potez

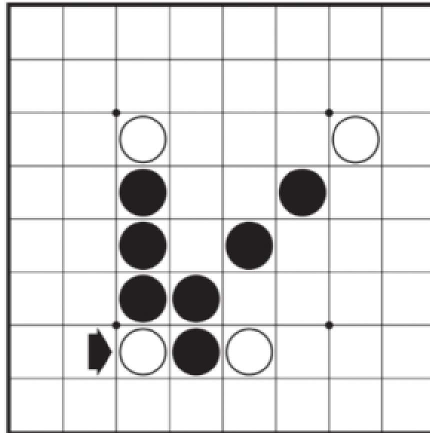
Svaki potez se sastoji od postavljanja jednog žetona svoje boje na ploču. Potez je valjan ako i samo ako će se njegovim odigravanjem okrenuti barem jedan žeton suprotne boje. Ako između novopostavljenog žetona i bilo kojeg drugog žetona iste boje kao novopostavljeni postoje samo žetoni suprotne boje bez ikakvih praznina između (u bilo kojem od 8 smjerova; horizontalno, vertikalno i dijagonalno), svi žetoni između ta dva bit će okrenuti (i samim time postati iste boje kao i novopostavljeni žeton). Na slici 2.2 prikazani su svi mogući potezi iz početne konfiguracije. Na slikama 2.3 i 2.4 prikazano je odigravanje poteza.



Slika 2.2: Svi mogući potezi crnog igrača na početnoj konfiguraciji



Slika 2.3: Postavljanjem žetona označenog strelicom okreće se samo disk 1



Slika 2.4: Postavljanjem žetona na mjesto označeno strelicom okrenut će se svi crni diskovi

2.1.2. Nemogućnost igranja poteza

Ako igrač koji je na potezu ne može odigrati nijedan valjani potez on jednostavno preskače taj krug i njegov protivnik je opet na potezu. Igrač na potezu mora odigrati neki valjani potez ako takav postoji bez obzira koliko loš bio taj potez. Ako se na svim poljima nalaze žetoni ili ako nijedan igrač ne može odigrati valjani potez igra završava.

2.1.3. Odabir pobjednika

Pobjednik je onaj igrač koji ima više žetona svoje boje na ploči u trenutku završetka igre. Ako se na ploči nalaze 32 bijela i 32 crna žetona igra se proglašava neriješeno.

2.1.4. Implementacija

Ploča je implementirana kao dvodimenzionalni niz veličine 8x8 tipova *Piece* u kojem je crni žeton označen s -1, bijeli s 1 te prazno polje s 0.

```
enum Piece : int8_t {
    Black = -1,
    None = 0,
    White = 1
};
```

Slika 2.5: Tip *Piece*

3. Neuronska mreža

Kako bi procijenili koliko je dobar neki potez odnosno stanje ploče do kojeg bi dovelo odigravanje tog poteza koristit ćemo neuronsku mrežu. U ovom radu korištena su 3 modela neuronske mreže različite kompleksnosti koji će biti pojašnjeni u nastavku.

3.1. Umjetni neuron

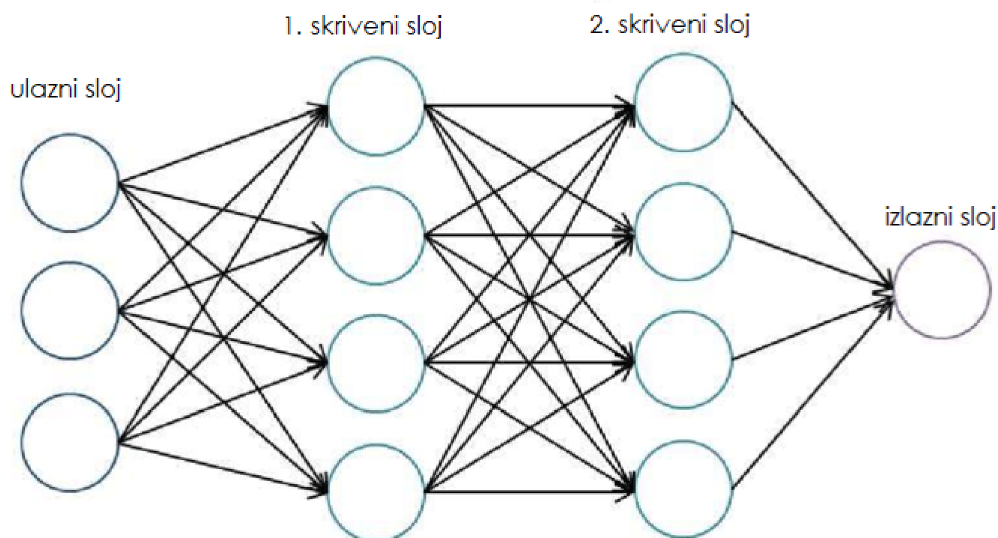
Umjetni neuron matematička je funkcija temeljena na modelu biološkog neurona te je on temeljni procesni element neuronske mreže[2]. Umjetni neuron funkcija je oblika:

$$y = \phi(\sum_{j=0}^{m} w_j * x_j) \tag{3.1}$$

pri čemu je $x_0 = 1$ poznato kao pristranost (eng. *bias*), x_1 do x_m ulazne vrijednosti, w_0 do w_m težine (eng. *weights*) te ϕ aktivacijska funkcija. Vrijednost izlaza neurona dalje se propagira kao ulazna vrijednost idućeg sloja neuronske mreže.

3.2. Unaprijedna neuronska mreža (eng. *feedforward*)

Neuronska mreža skup je međusobno povezanih neurona. Najčešće je prikazana grafom s usmjerenim vezama. Unaprijedna neuronska mreža je mreža u kojoj usmjerene veze između čvorova ne formiraju nijedan ciklus. Neuroni se dijele na ulazne, skrivene i izlazne. U ulazne neurone ne ulazi nijedna veza dok iz izlaznih ne izlazi nijedna veza. Mreža se najčešće dijeli na nekoliko slojeva: jedan ulazni i izlazni sloj te proizvoljan broj skrivenih slojeva između. Valja napomenuti da unaprijedne neuronske mreže nemaju svojstvo pamćenja jer informacija uvijek ide u sljedeći sloj odnosno nikad se ne vraća. Na slici 3.1 prikazana je unaprijedna neuronska mreža s dva skrivena sloja.



Slika 3.1: Primjer unaprijedne neuronske mreže s dva skrivena sloja

3.3. Linearni model

Prvi model napravljen je kao unaprijedna neuronska mreže bez skrivenog sloja. Na ulaznom sloju nalazi se 12 čvorova. Prvih 10 koristi se za statičku analizu ploče, 1 za kažnjavanje slobode igrača na ploči i 1 za nagrađivanje zauzimanja žetona na rubovima. Na izlaznom sloju nalazi se samo 1 čvor, ocjena stanja ploče. U daljnjim potpoglavljima detaljnije su pojašnjeni spomenuti pojmovi.

3.3.1. Statička analiza ploče

Valja primijetiti da je ploča simetrična s obzirom na horizontalnu, vertikalnu i dijagonalne osi. Uz naizgled valjanu i logičnu pretpostavku da su svi dijelovi na koje te osi dijele ploču jednako važni evaluaciju ploče može se iz početna 64 ulazna čvora sažeti na svega njih 10. Ta podjela detaljnije je prikazana na slici 3.2.

3.3.2. Ocjena slobode igrača

Sloboda žetona definira se kao broj praznih polja neposredno pored njega (uključujući i dijagonalno). Sloboda igrača je zbroj sloboda svih njegovih žetona. Ocjena slobode igrača definira se kao razlika slobode igrača i slobode njegovog protivnika[10]. Ideja iza ove metrike je da što je veća sloboda igrača to njegov protivnik ima više mogućih poteza.

Slika 3.2: Statička evaluacija ploče pomoću 10 parametara

3.3.3. Nagrađivanje zauzetih kutova

Ako se žeton nalazi u bilo kojem kutu on više nikad neće moći biti okrenut te se zbog toga "kutni" žeton smatra najvrijednijim žetonom. Isto svojstvo imaju svi njegovi susjedni žetoni iste boje koji se nalaze na rubu ploče. Očito je da takvi žetoni donose neku dobit te da ih treba nagraditi. Dodatno ako igrač postavi žeton u bilo koje od 3 susjedna polja nekom kutu postoji opasnost da će njegov protivnik u idućem koraku zauzeti taj kut, no ukoliko je taj kut već zauzet utoliko te opasnosti nema. Iz tog se razloga vrijednost susjednih polja kuta u statičkoj evaluaciji postavlja na 0 ako je taj kut već zauzet. Na slici 3.3 dan je primjer sloboda žetona i nagrađenih žetona.

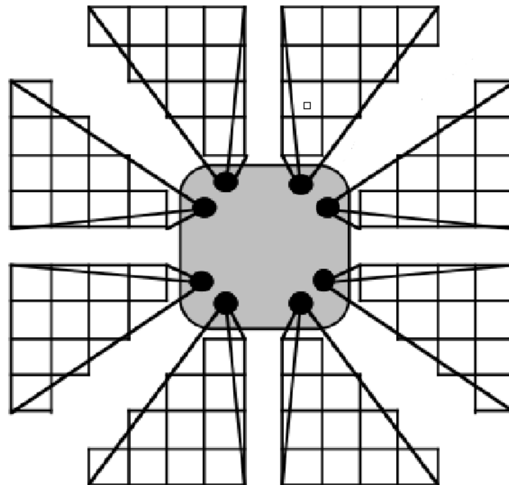
Slika 3.3: Sloboda bijelog igrača i nagrađeni žetoni susjedni kutnom

3.4. Drugi model

U drugom modelu korišten je drugačiji pristup koji se ne fokusira na poznata korisna svojstva već ih pokušava sam naučiti. U neuronskoj mreži nalazi se jedan ulazni sloj, 2 skrivena sloja te jedan izlazni sloj.

3.4.1. Ulazni sloj

Glavna ideja je da ploču podijelimo na trokute kao na slici 3.4 te svaki trokut evaluiramo u neku vrijednost koju propagiramo u idući sloj. Svaki trokut sastavljen je od 10 polja, a na ploči postoji 8 trokuta. Ovakva implementacija bi u ulaznom sloju imala 80 čvorova koji bi se u idući sloj propagirali pomoću 88 težina, no možemo kao u prošlom modelu pretpostaviti da su svi trokuti jednako važni te za sve trokute koristiti iste težine. Tako bi se broj težina iz ulaznog u prvi skriveni sloj smanjio s 88 na 11.



Slika 3.4: Prikaz ulaznog i prvog skrivenog sloja preuzeto iz [12]

3.4.2. Skriveni i izlazni sloj

U prvom i drugom skrivenom sloju nalazi se po 8 čvorova koji su potpuno povezani sa svim čvorovima idućeg sloja. U izlaznom sloju nalazi se samo jedan čvor koji predstavlja evaluaciju stanja ploče.

Kao aktivacijska funkcija korištena je sigmoidalna funkcija:

$$f(x) = \frac{1}{1 + e^{-x}} \tag{3.2}$$

U konačnici imamo neuronsku mrežu koja ima (10+1)+(8*(8+1))+(8+1)=92 težine. Implementiran je još jedan model koji koristi više struktura prikazanih na slici 3.4, ali je bio znatno sporiji i davao jednako dobre ili lošije rezultate kao prethodna dva modela.

3.5. Minimax algoritam

Za svako stanje ploče moguće je odrediti sve moguće ishode simulirajući sve moguće valjane poteze. Ako oba igrača imaju tu mogućnost oba će igrati optimalno te će ishod igre uvijek biti isti. Razlog zašto to nije moguće je taj što broj mogućih stanja raste eksponencijalno. Procjenjuje se da je broj mogućih stanja u igri Othello 10^{28} te da njeno stablo igre ima oko 10^{58} čvorova[5].

Ideja iza minimax algoritma je da svaki igrač želi maksimizirati svoj dobitak i minimizirati dobitak protivnika te će to napraviti ocjenjivanjem čvorova. Terminalno stanje definiramo kao stanje u kojem igra završava. Njemu se pridružuje neka vrijednost. Definira se heuristička funkcija koja na ulaz prima stanje ploče te na izlaz daje ocjenu tog stanja. Dodatno se definira maksimalna dubina k do koje će algoritam ići. Kad algoritam dođe do stanja na maksimalnoj dubini k umjesto da nastavi ići dublje te ako stanje nije terminalno vraća vrijednost heurističke funkcije. Ako u bilo kojem trenutku algoritam dođe do terminalnog stanja vraća njegovu vrijednost.

Sada je jednostavno primijeniti minimax algoritam odnosno ocijeniti neki čvor; ako se algoritam nalazi u čvoru u kojem je igrač na potezu uzima maksimalnu ocjenu njegove djece, a ako se nalazi u čvoru u kojem je njegov protivnik na potezu uzima minimalnu ocjenu njegove djece. Bitno je pripaziti da je u igri Othello moguće da isti igrač bude više puta zaredom na potezu (u slučaju da njegov protivnik nema valjanih poteza). Kao heuristička funkcija korištena je neuronska mreža opisana u prethodnim poglavljima.

3.5.1. Alfa-beta podrezivanje

Alfa-beta podrezivanje je optimizacija minimax algoritma koja iskorištava činjenicu da nije potrebno simulirati sve poteze maksimalne dubine k jer su neki potezi sigurno lošiji ili sigurno bolji od prethodno provjerenih poteza. Minimax algoritam dodatno pamti dvije vrijednosti α i β. α predstavlja najveću do sad viđenu ocjenu čvora u kojem je igrač na potezu, a β najmanju do sad viđenu ocjenu čvora u kojem je igračev protivnik na potezu. Ako u nekom trenutku vrijedi $\beta \leq \alpha$ izvođenje ocjenjivanja čvora

se može prekinuti jer α samo može rasti, a β samo može padati.

4. Genetski algoritmi

Genetski algoritmi pripadaju široj klasi algoritama zvanih evolucijski algoritmi koji su pak podskup evolucijskog računarstva. Genetski algoritmi su skup metaheuristika inspiriranih procesom prirodne selekcije te kao takvi koriste neke genetske operatore poput mutacija, križanja i selekcije. Najčešće se koriste u rješavanju optimizicijskih problema.

Princip na kojem rade je u suštini dosta jednostavan iterativan proces. Na početku se inicijalizira početna generacija koja se sastoji od određenog broja jedinki. Svaka jedinka predstavlja jedno moguće rješenje problema te se često naziva kromosom. Zatim se izračuna funkcija dobrote svake jedinke. Jednostavno rečeno ta funkcija jednim brojem predstavlja koliko je jedinka "dobra" u kontekstu problema koji se rješava. Zatim se primjenom operatora selekcije, križanja i mutacija iz stare generacije stvori nova generacija jedinki. U daljnjim potpoglavljima detaljnije je pojašnjen svaki ovaj pojam i prikazan u kontekstu imlementacije ovog rada.

4.1. Operator križanja

Kako bi iz jedinki stare generacije dobili neku novu jedinku potrebno je na neki način kombinirati neke kromosome kako bi dobili novi. To činimo operatorom križanja. Operatori križanja najčešće se dijele u sljedeće 3 kategorije ovisno o broju roditeljskih kromosoma:

- aseksualne - kromosom generiran iz jednog roditelja
- seksualne - iz dva roditelja se generira jedan ili dva kromosoma
- multi-rekombinacijske - iz više od dva roditelja se generira jedan ili više kromosoma

4.1.1. Simulirano binarno križanje (SBX)

Simulirano binarno križanje seksualni je operator križanja koji je nastao s ciljem primjene "one-point" križanja na realne (one u kojima su svojstva kromosoma prikazana realnim brojevima) reprezentacije kromosoma. Taj operator iz roditelja $x_1(t)$ i $x_2(t)$ generira djecu $\bar{x}_1(t)$ i $\bar{x}_2(t)$ prema sljedećoj formuli:

$$\bar{x}_{1j} = 0.5[(1 + \gamma_j)x_{1j}(t) + (1 - \gamma_j)x_{2j}(t)] \tag{4.1}$$

$$\bar{x}_{2j} = 0.5[(1 - \gamma_j)x_{1j}(t) + (1 + \gamma_j)x_{2j}(t)] \tag{4.2}$$

pri čemu je

$$\gamma_j = \begin{cases} (2r_j)^{\frac{1}{\eta+1}} & ako\ je\ r_j \leq\ 0.5 \\ (\frac{1}{2(1-r_j)})^{\frac{1}{\eta+1}} & \text{inače} \end{cases} \tag{4.3}$$

$r_j \sim U(0, 1)$, a $\eta > 0$ distribucijski indeks. Autori ovog operatora sugeriraju uporabu vrijednosti $\eta = 1$[6].

4.2. Operator selekcije

Kod operatora križanja nismo se dotakli teme odabira roditelja. Tome služi operator selekcije.

4.2.1. Turnirska selekcija

Jedan od najčešćih operatora selekcije upravo je turnirska selekcija koja je pogodna zbog svoje jednostavnosti i male složenosti. Iz populacije se nasumično odabere $n_t < n$ jedinki pri čemu je n broj jedinki unutar populacije jedne generacije te se uzme najbolja jedinka među odabranima. Korisno svojstvo ovog operatora je lakoća mijenjanja selekcijskog pritiska. Povećanjem parametra n_t povećava se i selekcijski pritisak. U ovom radu korišten je $n_t = 2$ koji ima relativno niski selekcijski pritisak te je kombiniran s elitizmom - metodom koja određeni broj najboljih (onih koji imaju najveću vrijednost funkcije dobrote) jedinki iz stare generacije sačuva odnosi prebaci u novu generaciju bez ikakvih promjena.

4.3. Operator mutacije

Cilj mutacije je unijeti novi genetski materijal u već postojeće jedinke te time izbjeći "zapinjanje" u lokalnim ekstremima. Naime, operatori križanja ne donose nikakav

novi genetski materijal jer oni samo kombiniraju već postojeće jedinke.

U radu je korištena metoda u kojoj u slučaju odabira jedinke za mutiranje svaki njen parametar ima vjerojatnost

$$p = \frac{1.5}{n_p} \tag{4.4}$$

da bude mutiran pri čemu je n_p broj parametara mreže koji ovisi o korištenom modelu. Mutiranje se vrši tako da se na originalnu vrijednost parametra doda Gaussov šum.

4.4. Dinamičke mutacije i križanja

Jedan veliki problem određivanje je koliko jedinki mutirati i križati unutar svake generacije. Primjerice unutar manjih populacija možda je povoljnije češće koristiti operator mutacije zbog manje raznolikosti jedinki unutar generacije. Još jedna moguća pretpostavka je da primjerice u 1. generaciji nije dobro koristiti isti broj mutacija kao u 1000. generaciji iz razloga što u 1. generaciji generalno postoji puno veća raznolikost nego u 1000. Kao rješenje tog problema korištena je varijacija na DHM/ILC (*Dynamic Decreasing of High Mutation Rate/Increasing of Low Crossover Rate*) pristup za veće populacije i varijacija na ILM/DHC (*Dynamic Increasing of Low Mutation/Decreasing of High Crossover*) pristup za manje populacije jer su pokazali dobre performanse za iste[9]. Kod DHM/ILC pristupa vrijed sljedeće:

$$p_c = \frac{trenutnaGeneracija}{ukupnoGeneracija} \tag{4.5}$$

$$p_m = 1 - p_c \tag{4.6}$$

Svaka jedinka unutar generacije imat će vjerojatnost p_c da bude zamijenjena novom jedinkom dobivenom križanjem te vjerojatnost p_m da bude mutirana. Vjerojatnost p_c linearno se povećava dok se vjerojatnost p_m linearno smanjuje. Kod ILM/DHC pristupa vrijedi:

$$p_m = \frac{trenutnaGeneracija}{ukupnoGeneracija} \tag{4.7}$$

$$p_c = 1 - p_m \tag{4.8}$$

4.5. Funkcija dobrote

Funkcija dobrote služi za ocjenu kvalitete jedinke. Problem je što u igri Othello ne postoji potpuno objektivna ocjena te kvalitete. Prva ideja je da se jedinke unutar jedne generacije natječu međusobno, no problem s tom idejom je da su kretnje jedinki potpuno

determinističke jer koriste samo neuronsku mrežu za odabir idućeg poteza. Konkretno dvije jedinke bi u međusobnom okršaju uvijek odigrale iste poteze. Kako bi se riješio taj problem u svakoj generaciji generiran je određen broj "početnih" konfiguracija ploče pomoću agenta koji igra nasumične poteze. Početnu konfiguraciju definiramo kao konfiguraciju u kojoj je popunjeno najviše pola ploče (odnosno odigrano je maksimalno 28 poteza).

Kako bi između dvije jedinke odredili onu bolju, odabire se 5 početnih konfiguracija ploče te svaka jedinka jednom po konfiguraciji igra kao crni te jednom kao bijeli. Pobjednik konfiguracije je ona jedinka koja ima veću sumu razlika svojih i protivničkih žetona u te dvije partije. Ako je ta suma razlika jednaka smatra se da su na toj konfiguraciji jedinke jednako dobre. Bolja jedinka je ona koja skupi više pobjeda. Ako jedinke imaju jednako pobjeda smatra se da su jednako dobre.

Kako bi odredili dobrotu jedinke ona se natječe s 50 nasumično odabranih jedinki iz svoje generacije te se boduje po sljedećem principu:

- +10 bodova ako je bolja od druge jedinke
- +1 bod ako je jednako dobra
- +0 bodova ako je lošija

Ideja iza ovakvog bodovanja je da se pobjede puno više nagrađuju od bivanja jednako dobrim. Što je veći broj konfiguracija na kojima se jedinke natječu i broj jedinki koje se natječu međusobno to bi ocjena dobrote jedinki trebala biti bolja po cijeni sporijeg izvršavanja programa.

Jedan potencijalni problem ovog pristupa je to što nije objektivan jer se jedinke cijelo vrijeme natječu samo međusobno te se može dogoditi da samo cijelo vrijeme loše jedinke igraju protiv loših jedinki te će zapravo onda najbolja jedinka još uvijek jednostavno biti loša. U svrhu borbe protiv tog problema svakih x generacija će najboljih y jedinki odigrati veliki broj partija protiv igrača koji igra nasumične poteze (pola partija kao bijeli i pola kao crni). Tako će donekle objektivno moći biti prikazan napredak agenata.

5. Implementacija

Za implementaciju umjetne inteligencije i okoline u kojoj se ona trenira korišten je programski jezik C++. Uz to je dodatno implementirano korisničko sučelje u jeziku Python u kojem korisnik može igrati protiv neke istrenirane umjetne inteligencije iz rada. Za matrične operacije korištena je biblioteka *NumCpp* koja je ekvivalentna biblioteci *NumPy* u Pythonu.

5.1. Implementacija simulatora

Simulator je implementiran kao klasa *Simulator* koja sadrži ploču, podatke o tome tko je na potezu, je li igra gotova te dva pokazivača na podatak tipa *Generic_bot* koji predstavlja nadklasu svih agenata. Ona implementira sve funkcionalnosti igre: odigravanja poteza, provjere valjanosti poteza, provjere kraja igre i određivanja pobjednika.

5.2. Klasa *Generic_bot*

Klasa *Generic_bot* apstraktna je klasa koja implementira osnovne funkcionalnosti potrebne svim agentima od kojih su najbitnije:

- *get_all_moves* - vraća vektor koji sadrži sve moguće poteze za igrača zadane boje
- *play_move* - simulira odigravanje poteza na zadanoj ploči
- *get_next_move* - apstraktna funkcija koja za zadanu ploču vraća odabrani potez agenta
- *get_game_result* - vraća razliku broja bijelih i crnih žetona
- *does_move_exist* - provjerava postoji li valjani potez koji igrač može odigrati

Detaljniji prikaz klase vidi se na slici 5.1.

```
class Generic_bot{
private:
    bool is_white;
    double fitness;
public:
    void set_is_white(bool is_white) { this -> is_white = is_white; }
    void set_fitness(double fitness) { this -> fitness = fitness; }
    bool get_is_white() { return is_white; }
    double get_fitness() { return fitness; }
    void increase_fitness(double x) { fitness += x; }

    int distance(int x1, int y1, int x2, int y2);
    int get_game_result(const Board &board);
    bool is_move_legal(const Board &board, Move &move);
    bool is_out_of_bounds(int x, int y);
    bool does_move_exist(const Board &board, bool is_white);
    std::vector<Move> get_all_moves(const Board &board, bool is_white);
    void play_move(Board &board, Move &move);

    friend bool operator <(const Generic_bot &A, const Generic_bot &B) { return A.fitness < B.fitness; }

    virtual Move get_next_move(const Board &board, bool is_white) = 0;
    virtual Move get_next_move(const Board &board) { return get_next_move(board, is_white); }
};
```

Slika 5.1: Deklaracija klase *Generic_bot*

5.3. Klasa *GA_bot*

Ova klasa nasljeđuje klasu *Generic_bot* te služi kao bazna klasa svih modela opisanih u radu. Tri najbitnije funkcionalnosti koje ona implementira su:

– *get_next_move* - vraća potez koji će agent odigrati

– *evaluate_board_state* - jedina funkcija koju moraju implementirati sve podklase, vraća ocjenu stanja ploče za zadanog igrača

– *alpha_beta_pruning* - implementira alfa-beta podrezivanje

Jedini atributi klase su težine neuronske mreže i dubina alfa-beta podrezivanja. Klasa dodatno implementira funkcionalnosti spremanja i učitavanja neuronske mreže u i iz datoteke i postavljanje težina neuronske mreže. Detaljnije na slici 5.2. Ovu klasu nasljeđuju klase *GA_linear_bot*, *GA_NN_bot1* i *GA_NN_bot2* koje implementiraju ranije opisane neuronske mreže. Ispostavilo se da trenutnom implementacijom nije isplativo trenirati agente na dubini većoj od 2 zbog velikog vremena izvšavanja programa.

5.4. Klasa *Trainer*

Klasa *Trainer* koristi se za treniranje agenata. Pri inicijaliziranju klase postavljaju se hiperparametri genetskog algoritma. Pri inicijalizaciji populacije svi parametri jedinki poprimaju vrijednosti iz normalne distribucije. Nadalje, tijekom cijelog procesa evolucije agenata svi parametri jedinki skalirani su tako da poprimaju vrijednost iz intervala

```
class GA_bot : public Generic_bot {
protected:
    std::vector<nc::NdArray<double> > weights;
    int MAX_DEPTH = 2;
public:
    GA_bot() : Generic_bot(){};
    nc::NdArray<double> &get_weights(int id) { return weights[id]; }
    std::vector<nc::NdArray<double> > &get_weights() { return weights; }
    void set_weights(const nc::NdArray<double> &weights, int id) { this -> weights[id] = weights; }
    void set_weights(const std::vector<nc::NdArray<double> > &weights) { this -> weights = weights; }

    int get_number_of_layers(){ return weights.size(); }
    void write_to_file(const std::string &path);
    void load_weights(std::fstream &stream);
    double alpha_beta_pruning(Board &board, bool is_white, double alpha, double beta, int depth);

    virtual Move get_next_move(const Board &board, bool is_white) override;
    virtual double evaluate_board_state(const Board &board, bool is_white) = 0;
};
```

Slika 5.2: Deklaracija klase *GA_bot*

$[-1, 1]$. Također nudi se mogućnost učitavanja populacije iz datoteke za slučaj da je tijekom procesa treniranja došlo do prekida. Pri računanju funkcije dobrote svih jedinke valja pripaziti na to da s obzirom da jedinke unutar populacije igraju jedna protiv druge ako prva jedinka pobijedi drugu to je isto kao da je druga izgubila od prve te se tom obzervacijom može prepoloviti broj odigranih partija. To je konkretno implementirano tako da se svaku generaciju implicitno generira graf u kojem se nalazi n čvorova stupnja k pri čemu je n broj jedinki unutar generacije te k broj protivnika protiv kojih se jedna jedinka natječe.

```
public:
    Trainer(int GENERATION_SIZE=300, int TOURNAMENT_SIZE=2, int TOTAL_NUMBER_OF_GENERATIONS=501, int NUMBER_OF_GAMES_PLAYED=50,
            int ELITIST_LIMIT=10);

    void initialize_generation();
    void mutate(std::vector<nc::NdArray<double> > &weights);
    void crossover();
    void train(bool load=false);
    void evaluate_generation();
    void play_out(int p1_id, int p2_id);
    void load_from_file();

    std::vector<Setup> generate_board_configurations(int num_of_configurations);
    int tournament_select(const std::vector<int> &other_parents);
    std::vector<nc::NdArray<double> > simulated_binary_crossover(const std::vector<int> &parents);
    double evaluate_bot(int id);
    double evaluate_bot_vs_random(int id);

    static void print_board(const Board &board);
};
```

Slika 5.3: Deklaracije metoda klase *Trainer*

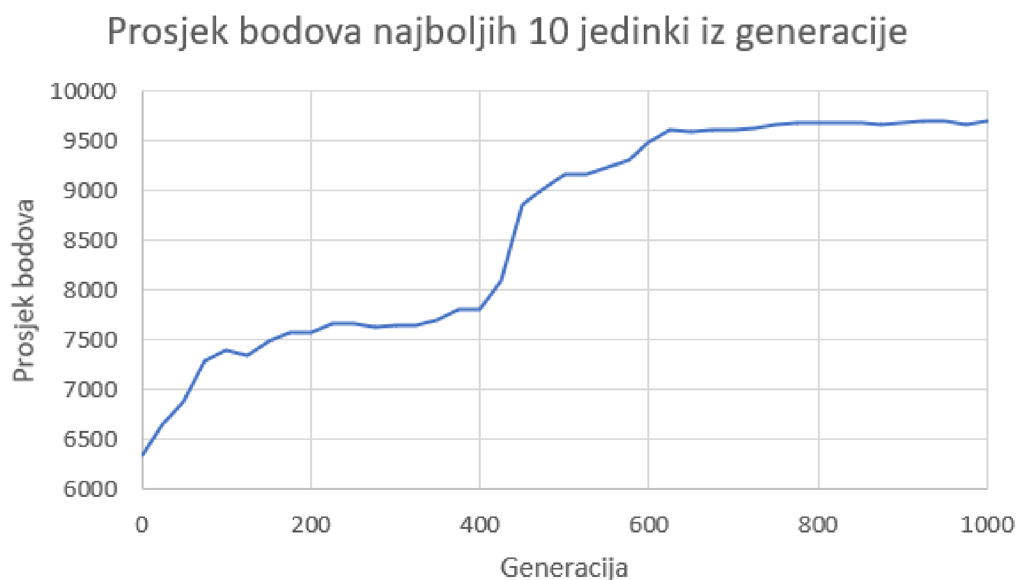
6. Rezultati

Napravljena je simulacija evolucije modela iz rada. U nastavku su prikazani rezultati evolucije uz korištenje dubine pretrage 1 i 2.

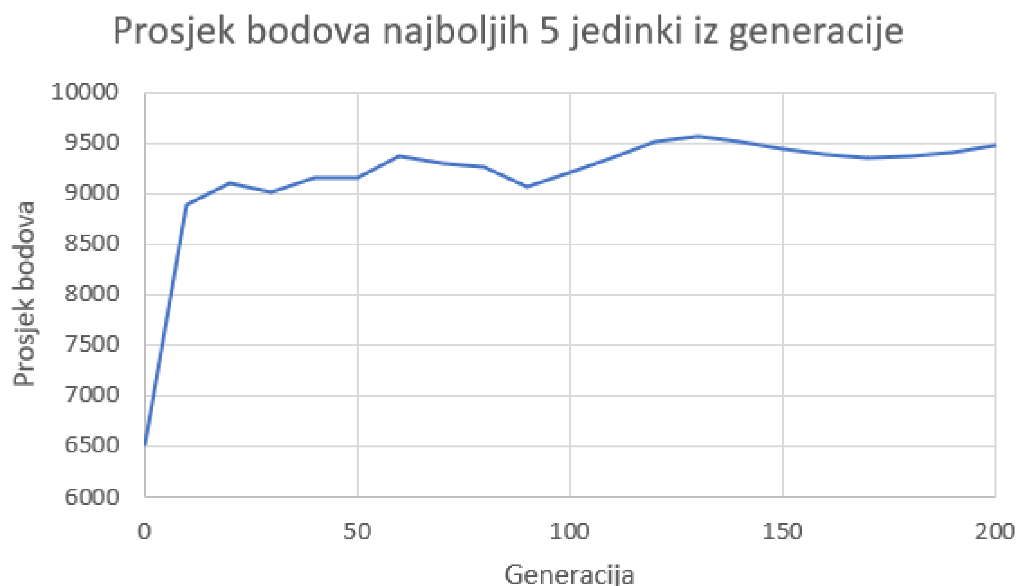
6.1. Model *GA_NN_bot2*

Za evoluciju uz dubinu pretrage jednog poteza korišteno je 1000 generacija te veličina populacije 300. Pri računanju funkcije dobrote svaka jedinka igra 10 partija (5 kao bijeli te 5 kao crni) protiv 50 drugih jedinki iz generacije. Pri prelasku u iduću generaciju sačuvano je 10 najboljih jedinki. Svakih 25 generacija najboljih 10 jedinki odigra 1000 partija protiv igrača koji igra nasumične poteze te je svaka pobjeda bodovana s 10 bodova, izjednačeno 1 bodom te poraz s 0 bodova. Vrijeme izvođenja jedne generacije iznosilo je oko 80 sekundi te je potpuna simulacija trajala oko 20 sati. Na slici 6.1 grafički su prikazani rezultati simulacije. Maksimalan mogući broj ostvarenih bodova svake jedinke bio je 10000 u ovom i svim narednim grafovima. U zadnjih 200 generacija prosjek bodova je stagnirao oko 9700 što je dosta visok rezultat.

Iako je prosječni faktor grananja igre Othello samo 10[11] te alfa-beta podrezivanje tu konstantu smanjuje, gledanje u dubinu još uvijek znatno usporava vrijeme izvođenja programa. Za evoluciju uz dubinu pretrage 2 korišteni su isti hiperparametri osim što je broj generacija smanjen na 200 te je veličina populacije smanjena na 100. Na slici 6.2 vidljivo je da jedinke na početku vrlo brzo napreduju te nakon dostignutog prosjeka iznosa oko 9500 dolazi do stagnacije. Valja napomenuti da što je prosjek bliže maksimalnoj vrijednosti to je teže napredovanje jedinki te je ova stagnacija trajala samo 50-ak generacija što nije pretjerano dug period. Nadalje, na manjoj dubini ovaj model je dosta sporo napredovao te bi za konkretnije rezultate trebalo na dubini 2 ispitati evoluciju na većem broju generacija.



Slika 6.1: Prosječni broj bodova najboljih 10 jedinki modela *GA_NN_bot2* u 1000 partija protiv igrača koji igra nasumične poteze pri dubini 1

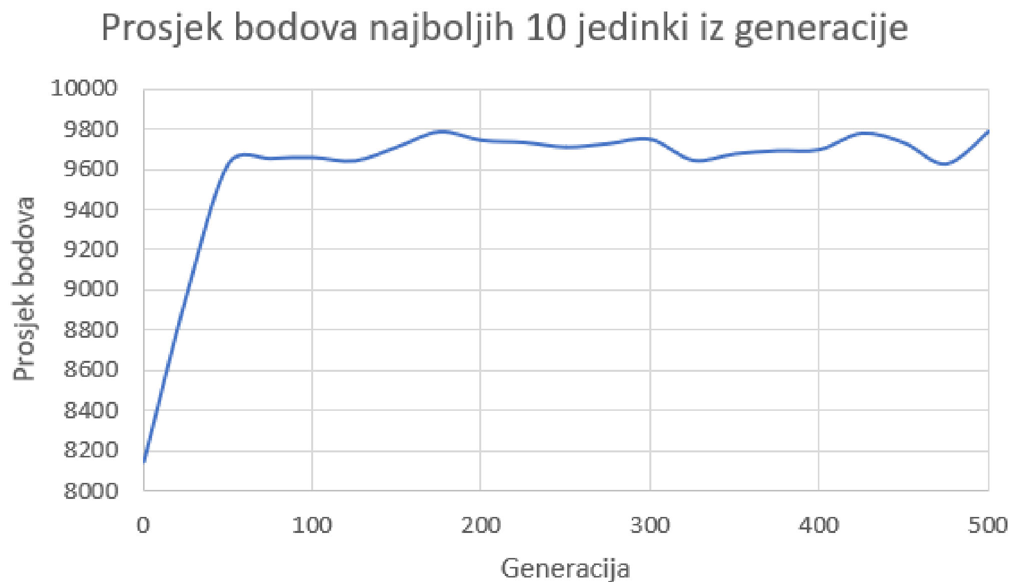


Slika 6.2: Prosječni broj bodova najboljih 5 jedinki modela *GA_NN_bot2* u 1000 partija protiv igrača koji igra nasumične poteze pri dubini 2

6.2. Model *GA_linear_bot*

Skoro identična simulacija napravljena je za linearni model iz poglavlja 3.3. Jedina je razlika što je korišten manji broj generacija (500) kod evolucije modela koji gleda 1

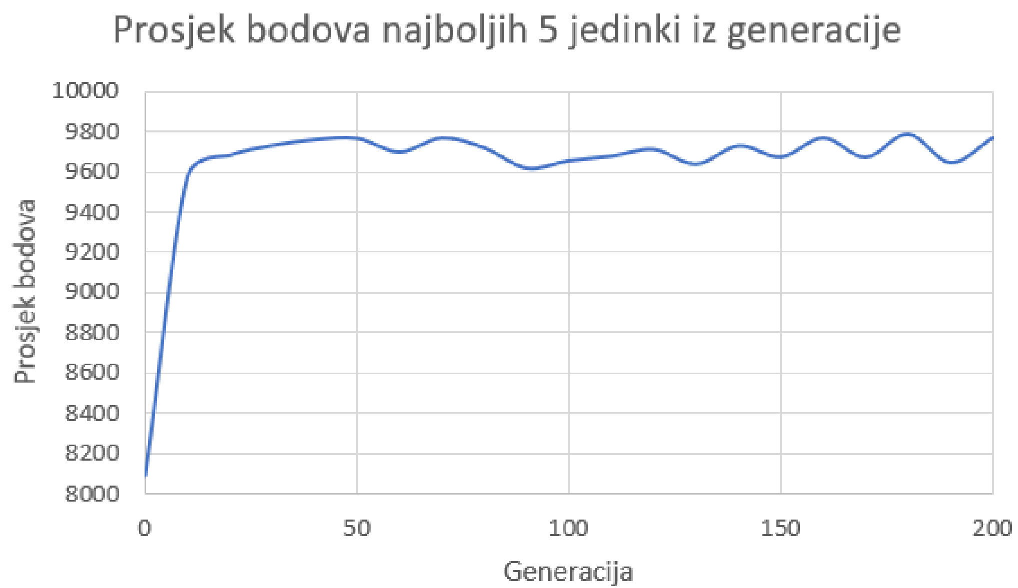
potez unaprijed iz razloga što je prosječni broj bodova brže stagnirao te je model znatno manje kompleksan od prethodnog. Model pri dubini 1 dostiže maksimalan prosjek bodova iznosa 9788 što je malo više od prošlog modela. Pri dubini dva model postiže otprilike iste rezultate kao i na dubini 1. Na obje dubine linearni model brže dolazi do dobrih rezultata što je i za očekivati zbog njegove značajno manje kompleksnosti. Zbog performansa prošlog modela na dubini 1 može se pretpostaviti da bi uz dulje treniranje on došao do boljih rezultata na dubini 2. Rezultati simulacije linearnog modela prikazani su na slikama 6.3 i 6.4



Slika 6.3: Prosječni broj bodova najboljih 10 jedinki modela *GA_linear_bot* u 1000 partija protiv igrača koji igra nasumične poteze pri dubini 1

6.3. Daljnji razvoj

Postoji mnoštvo opcija za poboljšanje rada u svim aspektima. Prvi je bolje traženje svih valjanih poteza i implementacija vraćanja poteza. Trenutno se svi valjani potezi traže tako da se pokuša postaviti žeton na svako prazno polje, no vjerojatno se mogu iskoristiti činjenice da se nakon svakog poteza skup novih valjanih poteza samo djelomično mijenja te da novi potezi mogu nastati samo u okolini novopostavljenog žetona. Nadalje, potreban je bolji način pregleda napretka agenata jer rezultati protiv igrača koji igra nasumične poteze variraju zbog njegove nasumičnosti. Također bi bilo dobro umjesto treniranja agenata međusobnim borbama pokušati ih trenirati protiv nekog



Slika 6.4: Prosječni broj bodova najboljih 5 jedinki modela *GA_linear_bot* u 1000 partija protiv igrača koji igra nasumične poteze pri dubini 2

"poznato dobrog" agenta. Osim toga treniranje agenata moglo bi se drastično ubrzati korištenjem više dretvi. Uz ove optimizacije vjerojatno bi bilo moguće pokušati trenirati agente na dubini 3 u nekom razumnom vremenu.

7. Zaključak

Ovaj rad istražio je primjenu genetskih algoritama na izradu agenta za igru Othello. Na početku rada pojašnjena su pravila igre. Zatim su objašnjene neuronske mreže, alfa-beta podrezivanje i genetski algoritmi te su opisani konkretni detalji modela i metoda korištenih u radu. Nakon toga prikazani su neki implementacijski detalji i rezultati.

Razvijena su dva modela: linearni model male kompleksnosti i složeniji model koji ploču rastavlja na trokute. Ispitane su mogućnosti treniranja modela na različitim dubinama te je linearni model pokazao malo bolje rezultate uz bržu konvergenciju koja je i bila očekivana s obzirom na drastično manji broj parametara. Složeniji model pokazao je lošije rezultate i dosta sporiju konvergenciju zbog kompleksnosti mreže. Oba modela naišla su na problem objektivne procjene njihovih performansi.

LITERATURA

[1] Alpha-beta pruning. URL https://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning. pristupljeno 5. lipnja 2023.

[2] Artificial neuron. URL https://en.wikipedia.org/wiki/Artificial_neuron. pristupljeno 28. svibnja 2023.

[3] Feedforward neural network. URL https://en.wikipedia.org/wiki/Feedforward_neural_network. pristupljeno 29. svibnja 2023.

[4] Genetic algorithm. URL https://en.wikipedia.org/wiki/Genetic_algorithm. pristupljeno 2. lipnja 2023.

[5] Louis Victor Allis et al. *Searching for solutions in games and artificial intelligence*. Ponsen & Looijen Wageningen, 1994.

[6] Kalyanmoy Deb i Ram Bhushan Agrawal. Simulated binary crossover for continuous search space. *Complex Syst.*, 9, 1995.

[7] Andries P Engelbrecht. *Computational intelligence: an introduction*. John Wiley & Sons, 2007.

[8] World Othello Federation. Othello rules. URL https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english. pristupljeno 28. svibnja 2023.

[9] Ahmad Hassanat, Khalid Almohammadi, Esra'a Alkafaween, Eman Abunawas, Awni Hammouri, i VB Surya Prasath. Choosing mutation and crossover ratios for genetic algorithms—a review with a new dynamic approach. *Information*, 10 (12):390, 2019.

[10] Nicolas Durand Jean-Marc Alliot. A genetic algorithm to improve an othello program. *Artificial Evolution, European Conference*, 1995.

[11] Kai-Fu Lee i Sanjoy Mahajan. Bill: a table-based, knowledge-intensive othello program. 1986.

[12] Anton Leouski. Learning of position evaluation in the game of othello, 1995.

[13] Co. Othello i Megahouse. Play othello online. URL https://www.eothello.com/. pristupljeno 28. svibnja 2023.

Razvoj agenta za igru Othello

Sažetak

Rad se bavi izradom agenta za igru Othello. Agent je implementiran korištenjem algoritma minimax i alfa-beta podrezivanja. Kao heuristička funkcija korištena je neuronska mreža trenirana genetskim algoritmom. Modeli su trenirani igranjem međusobno. Kao procjena koliko je model dobar odigran je velik broj partija protiv agenta koji igra nasumične poteze. Razvijena su dva modela različite kompleksnosti od kojih je jednostavniji pokazao malo bolje rezultate.

Ključne riječi: othello, umjetna inteligencija, genetski algoritam, neuronska mreža, minimax

Development of an agent for the Othello game

Abstract

This work deals with creating an agent for the game Othello. The agent is implemented using the minimax algorithm and alpha-beta pruning. A neural network trained with a genetic algorithm was used as a heuristic function. The models were trained playing versus themselves. To evaluate how good a model performs a large number of games were played versus an agent that moves randomly. As a result two models of different complexities were developed where the simpler one performed a bit better.

Keywords: othello, artificial intelligence, genetic algorithm, neural network, minimax