SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 7167

RAZVOJ STRATEGIJE ZA IGRANJE ŠAHA

Josip Kelava

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 7167

RAZVOJ STRATEGIJE ZA IGRANJE ŠAHA

Josip Kelava

Zagreb, lipanj 2021.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 12. ožujka 2021.

ZAVRŠNI ZADATAK br. 7167

Pristupnik:	**Josip Kelava (0036506017)**
Studij:	Računarstvo
Modul:	Programsko inženjerstvo i informacijski sustavi
Mentor:	doc. dr. sc. Marko Đurasević

Zadatak: **Razvoj strategije za igranje šaha**

Opis zadatka:

Proučiti problem razvoja strategije za igranje šaha, odnosno strategije za odabir prikladnog poteza na temelju trenutnog stanja šahovske ploče. Opisati najčešće metode iz literature koje su korištene za razvoj strategija za igranje šaha. Predložiti metodu za automatski razvoj strategije igranja šaha ili jednostavnije varijante iste igre. Ispitati učinkovitost razvijene metode, analizirati njezine prednosti i nedostatke te predložiti potencijalna poboljšanja za razvijenu metodu. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Rok za predaju rada: 11. lipnja 2021.

Sadržaj

1. Uvod

U današnje vrijeme umjetna inteligencija prisutna je u životima gotovo svih ljudi, u većoj ili manjoj mjeri – samim korištenjem računala često dobivamo sadržaj prilagođen nama. S vremenom, umjetna inteligencija pojavljuje se na sve više mjesta, upravljajući raznim sustavima i uvodeći automatizaciju u razne poslove.

Postoji mnogo različitih implementacija umjetne inteligencije – inicijalno su se implementirali algoritmi za deterministično rješavanje jasno definiranih i relativno lagano rješivih problema. S vremenom su se počeli implementirati i algoritmi za rješavanje zadataka koje nije moguće riješiti direktnim izračunom – bilo zato što je potrebno previše operacija za izračunati točno rješenje problema s dostupnim sklopovljem u danom vremenu, ili zato što ne postoji 'točno' rješenje. Za takve probleme potrebna su heuristička rješenja. Danas se češće korištenija rješenja razvijaju pomoću evolucijskog algoritma i/ili neuronskih mreža.

Igre su odlično okruženje za razvoj, inovaciju i testiranje novih ideja. Omogućuju simulaciju primjene nekog rješenja te ocjenu uspješnosti rješenja. Usporedimo to na primjer s programom za autonomnu vožnju automobila – čije testiranje zahtijeva implementaciju kompleksnih sustav emuliranja stvarnog svijeta, ili provedbu u stvarnom svijetu – nijedno trivijalno. Iz tog se razloga Deepmind, jedna od poznatijih tvrtki koja se bave umjetnom inteligencijom, bavi razvojem tehnologija izvedbi umjetne inteligencije na raznim igrama (šah, go, Starcraft II), nakon čega nove tehnologije koristi za daljnji napredak i istraživanje, ili ih pokušava iskoristiti za konkretne probleme u stvarnom životu (kao npr. simulacija proteina s ciljem nalaska cjepiva/lijeka za virus covid-19).

U ovom radu opisana je igra šah, minmax algoritam, neuronske mreže i evolucijski algoritam te implementacija istih.

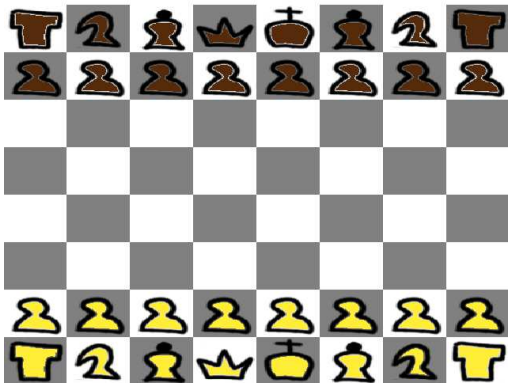
2. Igra „šah“

Šah je igra za dva igrača (bijeli i crni), nastala pred više od tisuću godina. Igrači naizmjence pomiču figure do kraja igre koji nastupa kad igrač na redu nema valjanih poteza, predajom, ili dogovorom o remiju (neriješeni rezultat). Prvi potez uvijek igra bijeli.

Na turnirima se rezultat jedne partije šaha obično boduje s jednim (1) bodom za pobjedu, pola (0.5) za neriješeno i nula (0) za poraz.

2.1. Početna pozicija

Šah se igra na šahovskoj ploči, koja se sastoji od 64 polja raspoređenih u 8 redaka (engl. *ranks*) i 8 linija (engl. *files*). Svako polje može se jednoznačno imenovati nazivom linije (jedno od slova a-h), te nazivom linije (broj 1-8). Početni raspored figura vidi se na slici 1.



Slika 1. Početna pozicija u šahu

Na prikazanoj ploči reci su raspoređeni od prvog do osmog od dolje prema gore, a linije od A do H s lijeva na desno. Tako su figure na ploči, u početnoj poziciji:

Topovi (kule, engl. *rook*) – bijeli na A1 i H1, crni na A8 i H8

Skakač (konj, engl. *knight*) – bijeli na B1 i G1, crni na B8 i G8

Lovac (engl. *bishop*) – bijeli na C1 i F1, crni na C8 i F8

Dama (kraljica, engl. *queen*) – bijela na D1, crna na D8

Kralj (engl. *king*) – bijeli na E1, crni na E8

Pješak (pijun, engl. *pawn*) – bijeli na A2-H2, crni na A7-H7

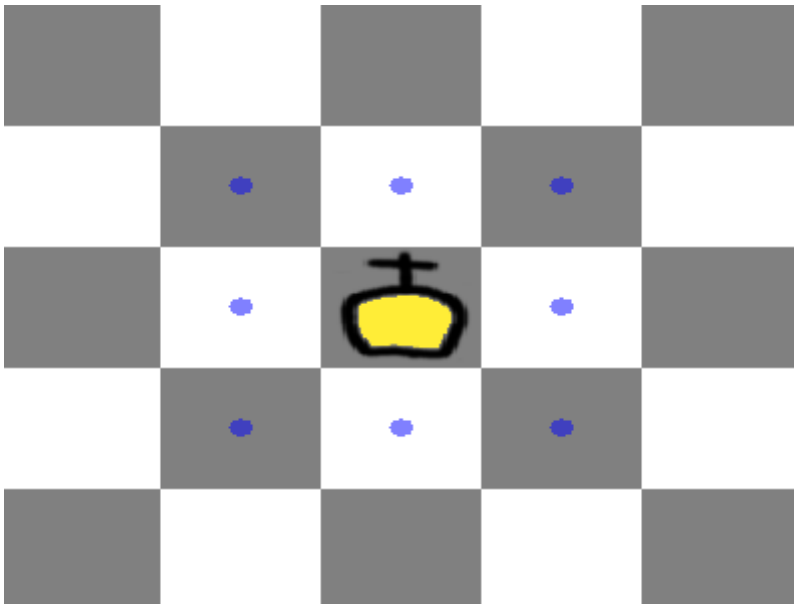
2.2. Kretanje i vrijednosti figura

Svaka od figura kreće se na unaprijed definiran način, te se njezina vrijednost može procijeniti u broju pješaka. Figura se nikad ne može pomaknuti na drugu figuru svoje boje, a ako se pomakne na figuru suprotne boje, ta se figura uklanja s ploče. Takav potez zove se uzimanjem figure (jedenjem, engl. *takes*).

Ako se figura kreće za više polja u nekom smjeru, ne smiju se nalaziti druge figure između početnog i završnog polja. Na kraju poteza, kralj igrača koji je igrao ne smije biti „napadnut“, tj. drugi igrač ne smije biti u mogućnosti uzeti protivničkog kralja.

2.2.1. Kralj

Kralj se u jednom potezu može pomaknuti za jedno polje u bilo kojem smjeru – po liniji, retku ili dijagonalno. Na slici 2 polja na koja kralj se može pomaknuti označena su plavim kružićima.



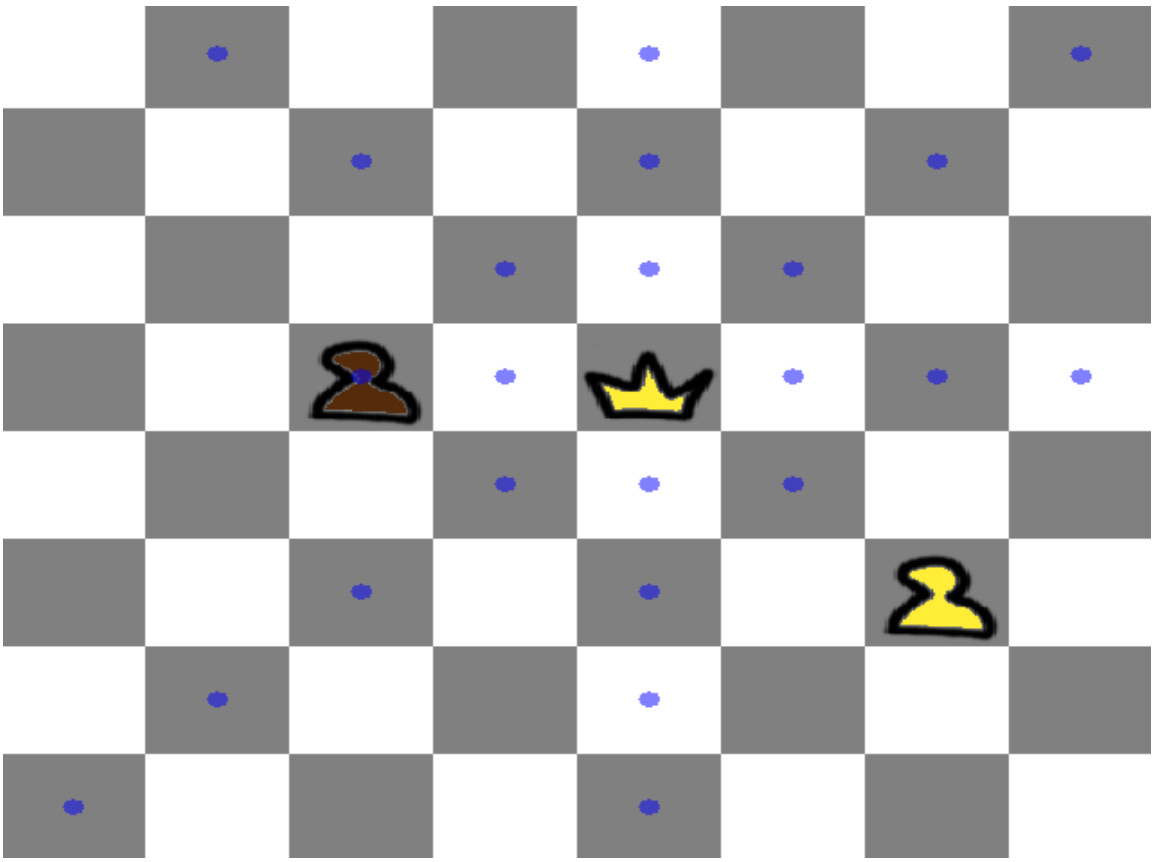
Slika 2. Kretanje kralja

2.2.2. Dama

Dama se u jednom potezu može pomaknuti za jedno ili više polja u bilo kojem smjeru.

Vrijednost dame se procjenjuje na 9 pijuna.

Na slici 3 vidi se kretanje dame po ploči, kao i kako figure blokiraju kretanje dame, topa i lovca.



Slika 3. Kretanje dame i blokiranje kretanja. Bijela dama može se pomaknuti na crnog pješaka i uzeti ga, ali ne i na bijelog.

2.2.3. Top

Top se u jednom potezu može pomaknuti za jedno ili više polja unutar linije ili retka u kojem se nalazi. Vrijednost topa procjenjuje se na 5 pješaka.

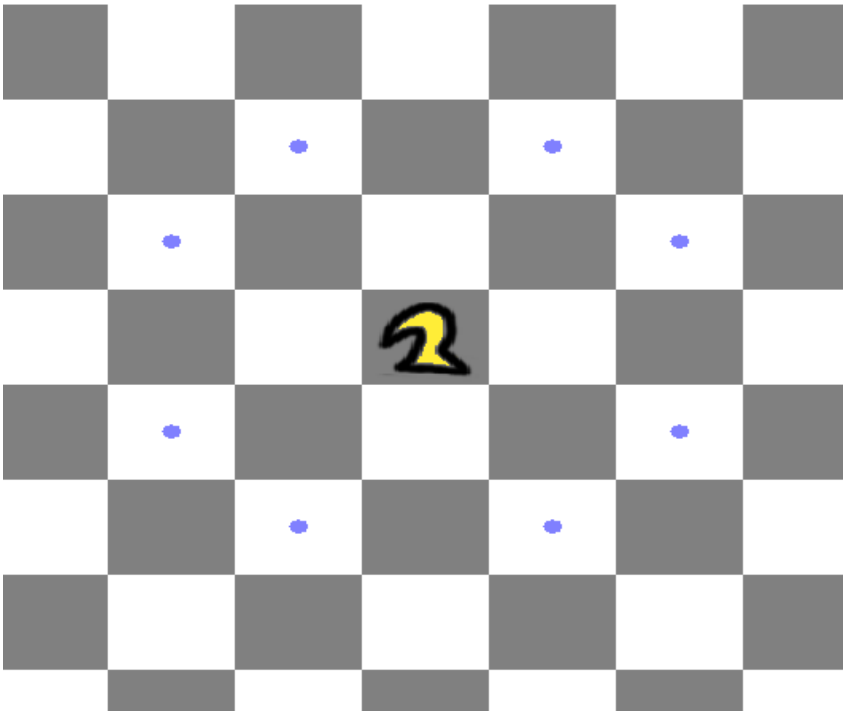
2.2.4. Lovac

Lovac se u jednom potezu može pomaknuti za jedno ili više polja dijagonalno. Procjena vrijednosti lovca je tri pješaka.

2.2.5. Skakač

Skakač se u jednom potezu može pomaknuti za tri polja u obliku slova „L“ – dva po liniji ili retku, zatim jedno u stranu (slika 4). Također, može „preskakati“ druge figure – nije bitno koje se figure nalaze na putu do polja, već samo je li konačno polje slobodno.

Vrijednost mu se procjenjuje na tri pješaka kao i lovcu, no općenito se smatra nešto malo slabijim.



Slika 4. Kretanje skakača

2.2.6. Pješak

Pješak se u jednom potezu može pomaknuti za jedno polje „unaprijed“ – bijeli na redak s većom oznakom ostajući u liniji na kojoj se nalazi, npr. s drugog retka na treći, dok se crni pomiče na redak s manjom oznakom. Takvo kretanje moguće je samo ako je to polje slobodno – pijun ne može uzimati na takav način. Ako pješak koji se pomiče još stoji na mjestu od početka igre, igrač može odlučiti pomaknuti ga i za dva polja unaprijed. Pješak može uzimati protivnikove figure, ali isključivo pomičući se dijagonalno unaprijed za jedno polje. Takvo dijagonalno kretanje moguće je samo uz uzimanje figure.

Ako pješak stane na zadnje polje (bijeli na osmi redak, crni na prvi), pješak se mijenja drugom figurom prema izboru igrača: u topa, lovca, skakača ili damu.

2.2.7. En passant (pješak)

Ako jedan igrač pomakne pješaka za dva polja, u trenutku kad bi pomicanje istog pješaka za jedno polje unaprijed omogućilo drugom igraču da uzme tog pješaka svojim pješakom,

drugi igrač idući i samo idući potez ima pravo uzeti tog pješaka svojim kao da je bio pomaknut za samo jedno polje.

2.2.8. Rohada (kralj, top)

Postoji još potez gdje se kralj miče za dva polja u stranu (u retku), te se kula istog igrača prema kojoj se igrač pomaknuo pomiče jedno polje preko kralja. Npr.: kralj s E1 na G1, top s H1 na F1. Da bi taj potez bio moguć, mora biti zadovoljen niz uvjeta:

- Kralj i top koji sudjeluje u rohadi ne smiju biti pomicani prije rohade
- Kralj, niti polje preko kojeg prelazi ne smije biti napadnuti
- Put između kralja i topa mora biti prazan

2.3. Završetak partije

Partija može završiti dogovorom između igrača (jedan igrač predaje partiju, ili jedan igrač predloži i drugi prihvati remi), ili zbog tijeka igre.

Igra na primjer završava kad igrač na redu ne može odigrati potez. Ako je kralj tog igrača napadnut, i ne može odigrati potez nakon kojega ne bi bio napadnut, to se smatra matom (šah-mat, engl. *chessmate*) te drugi igrač pobjeđuje. Ako kralj nije napadnut, radi se o patu (vrsta remija / neriješeno, engl. *stalemate*).

Postoji još par slučajeva gdje igra završava remijem (neriješeno):

- Ako ne postoji matna pozicija s figurama preostalim na ploči
- Nakon što se ista pozicija ponovi tri puta
- Ako se u zadnjih 50 poteza svakog igrača (ukupno 100) nije uzela niti jedna figura, ni pomaknuo niti jedan pješak

3. Teorija korištenih algoritama

U sklopu rada implementirana je umjetna inteligencija koja igra šah. Za implementaciju mreže korišteni su algoritmi minmax za odabir poteza i osnovnu procjenu pozicije, te neuronska mreža za dodatnu evaluaciji stanja pozicije. Također je implementiran i genetski algoritam koji omogućuje treniranje neuronskih mreža. U nastavku ovog poglavlja slijedu teoretska pozadina iza tih algoritama.

3.1. Minmax algoritam

Za neku poziciju moguće je odrediti sve moguće ishode, ovisno o odigranim potezima. Ako su oba igrača sposobna simulirati sve moguće poteze do kraja partije, mogu igrati savršeno i stoga se može odrediti kako će partija završiti iz bilo koje dane pozicije.

Broj mogućih pozicija raste eksponencijalno s brojem odigranih poteza, te današnja računala nisu sposobna simulirati sve moguće krajeve partija.

3.1.1. Koncept minmax algoritma

Dok nije moguće simulirati sve poteze do kraja partije, moguće je simulirati neki broj poteza od trenutne pozicije, ocijeniti sve moguće pozicije i na temelju toga odabrati potez. Ocjenu pozicije nije moguće odrediti točno kao pobjedu, remi ili poraz, umjesto čega se svakoj poziciji pridružuje neka brojčana vrijednost. Ta vrijednost se izračunava heuristički.

Za određenu poziciju izračunavamo njenu vrijednost kao izlaz heurističke funkcije ako se više ne simuliraju potezi iz te pozicije, ili kao najveća vrijednost pozicije u koju se dolazi ako igrač na redu odigra potez.

3.1.2. Pseudokod jednostavne minmax funkcije

```
MinMax(Pozicija, Dubina, MaxDubina):
        Ako (Dubina == MaxDubina):
                Vrati OcjeniPoziciju(Pozicija)
        NajvećaVrijednost = -Beskonačno
        NajboljiPotez = {}
        ZaSvakiMogućiPotez(Pozicija) Potez:
                Vrijednost = -MinMax(Pozicija.Odigraj(Potez)).Vrijednost
                Ako (Vrijednost > NajvećaVrijednost):
                        NajvećaVrijednost = Vrijednost
                        NajboljiPotez = Potez
        Vrati {Vrijednost = NajvećaVrijednost, Potez = NajboljiPotez}
```

Slika 5. Pseudokod minmax funkcije

U danom pseudokodu (slika 5) funkcije OcjeniPoziciju i minmax ocjenjuju poziciju iz perspektive igrača na potezu, gdje veća brojka znači da igrač na potezu ima veće šanse za pobjedu – veća brojka je povoljna za igrača na potezu.

Zbog toga što funkcije ocjenjuju poziciju iz perspektive igrača koji je na potezu, perspektiva se mijenja nakon svakog odigranog poteza stavlja se minus ispred rekurzivnog poziva minmax funkcije.

3.1.3. Optimizacija alfa-beta rezanja

Alfa-beta rezanje (engl. *alpha-beta pruning*) prisutno je u gotov svakoj implementaciji minmax algoritma. Sastoji se od rezanja rekurzivnih poziva minmax funkcije za pozicije koje ne mogu utjecati na rezultat konačnog rješenja.

Uzmimo u obzir neku poziciju A koju minmax ocjenjuje, te trenutno najbolju ocjenu CB(A), Iz pozicije A minmax se rekurzivno poziva za poziciju B, koja je nakon nekog broja daljnjih rekurzivnih poziva došla do trenutne najbolje vrijednosti CB(B). U trenutku kad se dogodi da je CB(B) >= -CB(A) može se prestati izračun ocjene za poziciju B. Vrijednost pozicije B može samo rasti, te nakon negacije sigurno neće utjecati na ocjenu pozicije A. To također vrijedi i za svaku poziciju s neparnim brojem poteza odigranih od pozicije A – odnosno, evaluacija neke pozicije može završiti na temelje bilo kojeg neparnog „roditelja“.

3.2. Genetski algoritam

Evolucija se, u stvarnom svijetu, pokazala kao jako moćan proces. Genetski algoritam implementira mehanizme evolucije – mutaciju, selekciju. Često implementira i križanje gena. Tok genetskog algoritma je opisan na slici 6.

```
Populacijia = GenerirajPočetnuPopulaciju()
Ponavljaj:
        OcjeniJedinke(Populacija)
        Populacija = GenerirajNovuPopulaciju(Populacija)
```

Slika 6. Pseudokod genetskog algoritma

3.2.1. Početna populacija

Početna populaciju može se generirati na više načina. Najčešća varijanta jest generiranje nasumične DNA (podaci koji definiraju jedinku / njeno ponašanje). Tako što obično počinje s lošim rezultatima te se s vremenom polagano poboljšava.

Alternativno, moguće je unaprijed konstruirati početne jedinke s definiranom početnom DNA s ciljem određenog početnog ponašanja jedinki.

Ukoliko se ponašanje jedinke definira i bez DNA, a DNA mijenja to ponašanje, moguće je generirati početnu DNA koja ne utječe na ponašanje jedinke što rezultira s osnovnim ponašanjem jedinke implementiranim neovisno o DNA.

3.2.2. Ocjena jedinki

Za svaku DNA u populacija konstruira se jedinka i/ili definira njeno ponašanje, te se ona testira simulacijom ponašanje jedinke u sustavu kojeg definira problem koji se rješava. Ako je moguće napraviti različite varijacije simuliranog sustava, simulacijom u nekoj specifičnoj varijaciji testirat će jedinke za različita svojstva nego neka druga varijacija. Odabir sustava može rezultirati pojavom različitih svojstava jedinki. To je korisno za dobiti različite, ili specijalizirane jedinke.

Bitno je da sve jedinke budu smještene u identični sustav. Ovisno o rješavanom problemu, simulacija može biti različitih priroda – jedinka može biti sama u simuliranom sustavu, ili s drugim jedinkama. Ako se jedinka smješta u sustav gdje se natječe s drugim jedinkama iz populacije, za svaki par jedinki populacije poželjno je da za svaku simulaciju unutar jedne

generacije u kojoj je jedna od tih jedinki sudjelovala postoji simulacija (moguće ista ta) u kojoj je sudjelovala druga jedinka i točno isti skup ostalih jedinki.

Ako se simulira sustav gdje se jedinke međusobno natječu, moguće je u sustav popuniti jedinkama populacije, ili jednom (testiranom) jedinkom iz populacije i unaprijed definiranom jedinkom čije ponašanje nikako ne ovisi o populaciji. Korištenjem neovisnih jedinki strogo definira svojstva sustava, a time i željena svojstva jedinki populacije. Stabilni zahtjevi također omogućuju mjerenje uspješnosti neke jedinke (i populacije). Međutim, imaju i negativnih posljedica – ako je jedinka populacije znatno lošija ili znatno bolja od fiksne jedinke, utjecaj mutacije na simulaciju gubi značaj što usporava evoluciju. Natjecanjem jedinki populacije protiv drugih jedinki iz populacije dobiva se pomična snaga zahtjeva, osiguravajući kontinuirani utjecaj mutacije na rezultate evolucije i uklanjajući meke granice evolucije zbog ograničenih izazova.

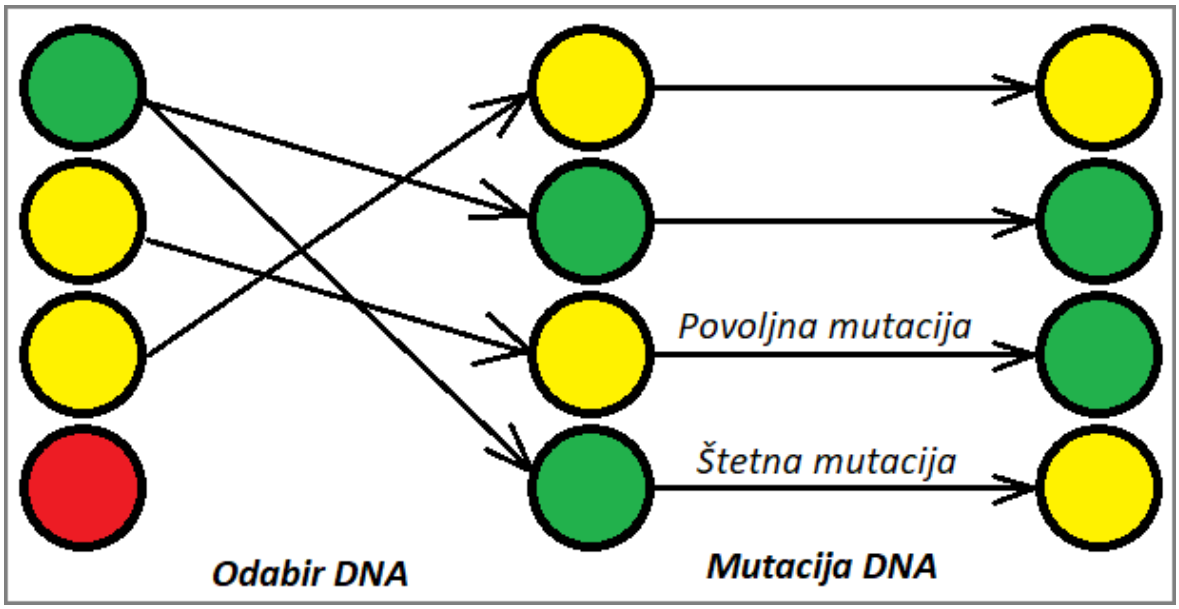
3.2.3. Prijelaz na iduću populaciju

Rezultati simulacije ocjenjuju se brojkom, koja opisuje uspješnost jedinke definirane testiranom DNA. Tu brojku zovemo funkcijom dobrote (engl. *fitness function*), i ona definira vjerojatnost pojavljivanja te DNA u sljedećoj generaciji.

Nova populacija puni se jedinkama blago izmijenjenih DNA jedinki iz prethodne generacije. Za svaku jedinku osnovna DNA (DNA iz prošle generacije koja će se promijeniti) nasumično se bira, gdje je šansa odabira neke DNA proporcionalna funkcije dobrote jedinke definirane tom DNA. Tako mutacije koje uzrokuju povoljno ponašanje dobiju veću dobrotu, imaju veće šanse propagacije u iduću generaciju. Slično, štetne mutacije rezultiraju manjom dobrotom, te se posljedično gube. Time dobivamo evoluciju, tj. postepeno poboljšanje ponašanja jedinki populacije.

Moguće je, pri izradi nove DNA, koristiti DNA dvije ili više jedinki prošle generacije, kombinirajući njihova svojstva (križanje, engl. *crossing*). Ako dvije jedinke imaju različita povoljna svojstva definirana različitim dijelovima DNA, moguće je križanjem zadržati oba svojstva.

U sustavima gdje se jedinke iz populacije međusobno natječu, bitno je zadržati raznolikost jedinki kako bi se dobili objektivniji zahtjevi simulacije. Ako sve jedinke postanu slične, generalno loše svojstvo se može pokazati dobrim u natjecanju s tom specifičnom vrstom jedinke, omogućujući širenje istog.



Slika 7. Pojednostavljeni prikaz generiranja nove populacije. Dobrotom mjerljiva kvaliteta DNA prikazana je bojom: lošija (crveno), srednja (žuto), bolja (zeleno). Spektar kvaliteta je relativan unutar ovog prelaska na iduću generaciju.

3.3. Neuronske mreže

Neuronska mreža je programska struktura koja oponaša strukturu mozga živih bića – sastoji se od skupa čvorova (neurona), povezanih usmjerenim vezama. U računalnom svijetu obično se konstruiraju kao usmjereni aciklički graf. Neuroni se dijele na ulazne neurone (u koje ne vodi ni jedna veza), unutarnje i vanjske (iz kojih ne vodi ni jedna veza). Struktura unutarnjih neurona je većinom podijeljena na slojeve, gdje veze uvijek povezuju neurone između dva susjedna sloja: iz ulaznih u prvi unutarnji, iz prvog u drugi, ... iz zadnjeg u izlazne.

3.3.1. Funkcionalnost neuronske mreže

Neuronska mreža pretvara niz podataka (brojeva) u novi niz brojeva. Vrijednosti ulaznih neurona postavljaju se na niz vrijednosti koje zadaje problem koji mreža rješava, vrijednosti ostalih čvorova u mreži izračunavaju se na temelju vrijednosti čvorova ulaznih veza. Vrijednosti izlaznih neurona predstavljaju odgovor na problem koji mreža rješava za neki ulazni niz. Funkcije vrijednosti neurona obično imaju kodomenu ograničenog intervala realnih brojeva, kao npr. hiperbolični tangens sume ulaza.

Tako na primjer, u simulaciji jedinke u nekom sustavu neuronska mreža može predstavljati mozak jedinke, gdje se vrijednosti ulaznih neurona postavljaju na svojstva sustava (koja jedinka „vidi“) u nekom trenutku, dok dobivene vrijednosti izlaznih neurona predstavljaju akcije jedinke u tom trenutku. Ili, ulazi mogu predstavljati boje piksela neke slike, a izlazi mogu reći radi li se o mački, psu, automobilu, ili čemu već. Neuronsku mrežu može se trenirati da radi puno različitih stvari.

Neuronsku mrežu može se trenirati na više načina, najčešće genetskim algoritmom i/ili algoritmom propagacija unatrag.

3.3.2. Propagacija unatrag (engl. *backpropagation*)

Ukoliko se za neki ulazni niz podataka može izraziti točnost izlaznih vrijednosti kao diferencijabilnu funkciju, moguće je promijeniti mrežu tako da se poveća točnost izlaza za neki ulazni niz, ili skupa ulaznih nizova.

U danom slučaju, moguće je izračunati diferencijal funkcije ovisno o vrijednosti bilo kojeg izlaznog čvora. Taj se diferencijal propagira unatrag kroz cijelu mrežu, jer vrijednost ne-ulaznih čvorova je funkcija vrijednosti povezanih čvorova – moguće je izračunati derivaciju funkcije točnosti ovisno o vrijednosti nekog čvora ako je poznata derivacija točnosti ovisno o vrijednostima svih čvorova prema kojima isti čvor ima vezu. Slično, za čvor koji je poznata derivacija, moguće je izračunati derivaciju funkcije točnosti ovisno o svojstvima tog čvora i svojstava veza koje vode u taj čvor. Nizom malih pomaka svih tih svojstava mreže ovisno o derivacijama moguće je povećati točnost mreže prema toj funkciji do nekog lokalnog maksimuma, ovisno o početnim svojstvima mreže.

3.3.3. Rekurentne neuronske mreže

Do sad opisane neuronske mreže nemaju nikakav način pamćenja – izlazni niz vrijednosti je funkcija isključivo ovisna o ulaznom nizu vrijednosti – za bilo ulazni niz uvijek se dobiva isti izlazni niz. Najčešći način implementacije pamćenja u neuronskim mrežama jesu rekurentne neuronske mreže.

Rekurentne mreže dodaju niz parova ulaznih i izlaznih čvorova. U prvoj evaluaciji neuronske mreže vrijednosti dodanih ulaznih čvorova se postavljaju na neke inicijalne vrijednosti, a u svakoj idućoj se postavljaju na vrijednost uparenog izlaznog neurona u prošloj evaluaciji.

4. Implementacija

Za implementaciju umjetne inteligencije korišten je programski jezik C++. Uz to implementirano je i grafičko sučelje koje omogućuje igranje šahovske partije protiv umjetne inteligencije u programskom jeziku Java.

Sve datoteke izvornog koda (engl. *source*) uključuju datoteku zaglavlja (engl. *header*) "StdH.h". Sve datoteke koje implementiraju main() funkciju smještene su unutar "Executable/..." direktorija, te podržavaju isključenje iz kompiliranja uz pomoć pretprocesorskih naredbi smještenih u datoteku zaglavlja "Executable/MainSwitch.h".

Izvorni kod za izvršnu datoteku koja komunicira s Java grafičkim sučeljem smješten je u "Executable/JavaAgent.cpp", a aktivira se preko makro-a `MAIN_USE_JAVA_UI`.

Kod je uglavnom formatiran prema standardima kodiranja Unreal Engine-a.

4.1. Implementacija šahovske ploče

Šahovska ploča implementirana je niz polja, od kojih svako pamti koja se figura nazali na polju. Figure su zapisane kao brojke, ali maskirane su kao enumeracija, vidljivo na slici 8.

```
enum class EChessPiece : int
{
    None = 0,
    WhitePawn = 1,
    WhiteKnight = 2,
    WhiteBishop = 3,
    WhiteRook = 4,
    WhiteQueen = 5,
    WhiteKing = 6,
    BlackPawn = -1,
    BlackKnight = -2,
    BlackBishop = -3,
    BlackRook = -4,
    BlackQueen = -5,
    BlackKing = -6
};
```

Slika 8. Enumeracija šahovskih figura.

Implementacija šahovske logike smještena je unutar klase `FChessBoard`, no dio logike ograničen je na bijele figure. Za pristup istoj logici za crne figure koristi se klasa `FDoubleBoard` koja sadrži dva člana klase `FChessBoard`. Jedna od tih ploča predstavlja ploču kakva jest, dok su na drugoj figure simetrično preslikane u odnosu na 4/5

redak i zamijenjenih boja. Na njoj početna pozicija izgleda identično kao i na pravoj ploči, osim toga što prvi potez vuče crni. Ta ploča koristi se da umjetna inteligencija igrala crnim figurama kao da su bijele – uklanja potrebu za dupliciranjem koda.

Šahovska ploča pamti koje su figure pomaknute koristeći bitmasku (jedno polje predstavljeno je jednim bitom) spremljenu kao 64-bitni integer (cijeli broj). Nakon što je figura pomaknuta na neko polje, označava se da je figura na tom polju pomaknuta. To je potrebno za praćenje uvjeta rohade. Slično, pamti se na koje polje je moguće odigrati potez *en passant*. Nakon svakog poteza *en passant* polje se briše, te se postavlja za svaki potez pješaka dva polja unaprijed. Zapis tog polja koristi se u pretrazi mogućih poteza pješaka.

Za potrebe minmax algoritma šahovska ploča također podržava vraćanje odigranih poteza unatrag tako da za svaki odigrani potez na stog dodaje podatke potrebne za vraćanje tog poteza:

- Za svako polje na kojem je prije poteza stajala druga figura nego nakon poteza, pamte se koordinate polja i stara figura (obično dva polja, tri za *en passant* i četiri za rohadu)
- *en passant* polje
- bitmasku polja pomaknutih figura

4.2. Implementacija minmax algoritma

Sve implementirane umjetne inteligencije implementiraju sučelje `IChessAI` čija je definicija prikazana na slici 9.

```
struct FEvaluatedMove
{
public:
    FEvaluatedMove(int MoveRankFrom = -1, int MoveFileFrom = -1,
        int MoveRankTo = -1, int MoveFileTo = -1, float BoardEval = 0.0f);
public:
    float Evaluation;
    int RankFrom;
    int FileFrom;
    int RankTo;
    int FileTo;

public:
    bool operator== (const FEvaluatedMove& Other) const;
    bool operator!= (const FEvaluatedMove& Other) const;
};

class IChessAI
{
public:
    virtual FEvaluatedMove ChooseMove(FDoubleBoard& Board) = 0;
    virtual bool PlayMove(FDoubleBoard& Board) = 0;
};
```

Slika 9. Definicija sučelja `IChessAI`

Minmax algoritam implementiran je u klasi `FMiMaxAI` u datoteci "MinMaxAI.h". Deklaracija klase vidljiva je na slici 10. Uz klasu definirane su i `float` vrijednosti `MINMAX_EVAL_WIN` i `MINMAX_EVAL_LOSS` kao brojevi dovoljno velika da budu strogo veći od bilo koje evaluacije pozicije. To je efektivno isto kao i beskonačno mali/velik broj, koji kao takav ne postoji u jeziku C++.

```
constexpr float MINMAX_EVAL_LOSS = -1e9f;
constexpr float MINMAX_EVAL_WIN = 1e9f;

class FMinMaxAI : public IChessAI
{
public:
    FMinMaxAI();
public:
    static float Eval(FDoubleBoard& Board, int NormalDepth, int VolatileDepth);

    virtual FEvaluatedMove ChooseMove(FDoubleBoard& Board) override;
    virtual bool PlayMove(FDoubleBoard& Board) override;
    void SetDepths(int Normal, int Volatile);
protected:
    virtual float Evaluate(FDoubleBoard& Board);
private:
    float MinMax(FDoubleBoard& Board, float Alfa, float Beta, int Depth, bool bVolatile);
public:
    FEvaluatedMove LastPlayedMove;
private:
    int MaxDepth = 4;
    int MaxVolatileDepth = 8;
};
```

Slika 10. Deklaracija klase `FMinMaxAI`

Funkcija PlayMove i ChooseMove su uglavnom funkcije omota koje pozivaju minmax funkciju te predaju njene rezultate. Implementacija minmax funkcije se uglavnom poklapa s pseudokodom danim u poglavlju 3.1.2. Evaluacija pozicije na ploči u vršnom pozivu minmax funkcije poziva se funkcija `Evaluate`. Ona računa razliku u vrijednostima figura na ploči igrača, te za svaku figuru dodaje još neku manju brojku ovisno o poziciji, npr. za pješake koliko su blizu srednjoj liniji te koliko su blizu promocije. Također, lovac je bodovan s 3.1 bodom umjesto 3, zbog čega umjetna inteligencija više cijeni lovca nego skakača.

Dodatno, podržan je dinamički maksimalni broj simuliranih poteza, ovisno o tome je li u prethodnom potezu uzeta neka figura – to je bitno jer, ako bude uzeta figura u zadnjem potezu, neće biti uzeto u obzir da figura koja uzima također može biti uzeta. Tako na primjer, bez tog dodatka umjetna inteligencija bojala bi se izgubiti branjenog (napadnutog od figure iste boje) pješaka od protivničke dame koja je znatno vrjednija, i probala izbjeći situaciju u kojoj je takvo uzimanje moguće.

Dodatno, pobjeda/poraz se ne detektiraju kao mat, već kao nedostatak kralja – što nije u duhu pravila šaha jer se kralja ne može uzeti, ali to je dovoljno dobro jer se tome pridaje „beskonačni“ prioritet pri evaluaciji pozicije tj. tretira se kao pobjeda/poraz, a zapravo je i ekvivalentno matu – mat jest pozicija u kojoj se ne može spriječiti protivnik da idući potez uzme kralja.

4.3. Implementacija neuronske mreže

Neuronska mreža implementirana je u klasi `FNetwork`, koja se sastoji od niza instanci klase `FNode` i niza pomoćnih podataka. Implementacija podržava proizvoljni broj grupa rekurentnih parova proizvoljne veličine. Klasa implementira veći broj javnih funkcija:

- `void FromDna(FDna& Dna)` – implementira konstrukciju neuronske mreže iz niza brojeva spremljenog u instanci klase `FDna` (deserijalizacija)
- `void ToDna(FDna& Dna)` – implementira pohranu oblika mreže u niz brojeva spremljenog u instancu klase `FDna` (serijalizacija). Takav niz se lako može pohraniti na disk i kasnije pročitati s njega, ili prenijeti nekom drugom programu. Funkcije mutacije su također implementirane tako da mijenjaju zapis DNA a ne samu mrežu, nakon čega se mreže rekonstruiraju iz mutirane DNA.

- `void SetInput(int Index, float Value)` – podešava vrijednost odabranog ulaznog neurona na odabrani realni broj
- `float GetOutput(int Index)` – vraća vrijednost odabranog izlaznog neurona
- `void Update()` – funkcija koja se poziva nakon postavljanja ulaznih vrijednosti, a prije čitanja izlaznih. Izračunava vrijednosti svih neurona u mreži (osim ulaznih), te prepisuje novo izračunate vrijednosti izlaznih rekurentnih čvorova u njihove ulazne parove.
- `void ResetRecurrent(int Level)` – resetira vrijednosti ulaznih rekurentnih neurona odabrane grupe na početne.

Izračun vrijednosti neurona implementiran je u klasi `FNode`. Ovisno o definiranim makroima u standardnom headeru (datoteka zaglavlja "StdH.h"), funkcija vrijednosti čvora može biti fiksna, ili preuzeta iz DNA. Dan je pseudokod izračuna na slici 11.

```
IzračunajNovuVrijednost(Čvor):
    NovaVrijednost = Čvor.Pristranost
    ZaSvakuUlaznuVezu(Čvor) Veza:
        UlaznaVrijednost = Veza.IzvorniČvor.Vrijednost * Veza.SnagaVeze
        NovaVrijednost = FunkcijaSabiranja(NovaVrijednost, UlaznaVrijednost)
    Čvor.Vrijednost = FunkcijaPreslikavanja(NovaVrijednost)
```

Slika 11. Pseudokod izračuna vrijednosti

Funkcije sabiranja odnosno preslikavanja definirane su u DNA iz koje se konstruira mreža ako su makroi `USE_CONSUMER_FUNCTIONS` odnosno `USE_MAPPING_FUNCTIONS` u standardnom headeru postavljeni na 1. Ako nisu, te funkcije su redom zbrajanje i hiperbolični tangens.

Implementirane su također funkcije koje omogućuju propagiranje unatrag, koje pretpostavljaju da su makroi iz prethodnog odlomka postavljeni na nulu, tj. da su funkcije svih čvorova hiperbolični tangens sume.

Ova klasa već sama po sebi prilično dobro i brzo igra šah na dubinama 4/8 (broj simuliranih poteza odnosno broj simuliranih poteza gdje je zadnji potez uzimanje figure), međutim ima prostora za poboljšanje, posebice potkraj partije – npr. nije sposobna matirati protivničkog kralja samo kraljem i damom, što je dosta jednostavno za izvesti.

4.4. Implementacija umjetnih inteligencija

Osim već spomenute klase umjetne inteligencije `FMinMaxAI`, implementirane su još dvije klase koje koriste neuronsku mrežu.

4.4.1. Klasa FNeuNetFullAI

Klasa `FNeuNetFullAI` koristi neuronsku mrežu sa 64 ulazna i 4 izlazna neurona, ne računajući rekurentne parove. Ulazne vrijednosti postavljaju se na brojčane vrijednosti figura na svakom polju kako su definirane u enuramciji `EChessPiece` – crne kao negativni brojevi, bijele pozitivni, a prazna polja točno nula. Vrijednosti izlaznih neurona predstavljaju redak i liniju polja s kojeg i na koje treba pomaknuti figuru ako se nalaze unutar intervala [0, tanh 0.8). Klasa podržava propagaciju unatrag – promjenu mreže prema gradijentu dobivenom za neku poziciju i željeni potez.

Uz tu umjetnu inteligenciju dolazi i klasa `FNeuNetFullMutator` koja implementira generaciju početne DNA, i kasnije mutaciju DNA u genetskom algoritmu.

Ova klasa međutim nije dala korisne rezultate. Šah je kompleksna igra, a ovakva implementacija podrazumijeva da neuronska mreža u potpunosti razumije igru i samostalno donosi odluke. To bi zahtijevalo veliku izrazito veliku neuronsku mrežu i posljedično znatno usporilo evaluaciju mreže i bilo kakvu evoluciju. Takav pristup mogao bi raditi, no uz visoke vremenske i hardverske zahtjeve. U testnom vremenu u sklopu ovog rada nije dobivena mreža koja bi mogla igrati šah, niti pojednostavljeni (na ploči 6x6 s manje figura i vrsta figura) – niti osnovnim genetskim algoritmom niti propagacijom unatrag.

4.4.2. Klasa FNetEvalMinMax

Također je implementirana klasa `FNetEvalMinMax` koja nasljeđuje klasu `FMinMaxAI`. Djeluje identično kao i bazna klasa, osim što se pri evaluaciji pozicije ploča predaje neuronskoj mreži u istom obliku kao i neuronskoj mreži umjetne inteligencije `FNeuNetFullAI`, ali umjesto poteza koji treba odigrati, čita se vrijednost jedinog izlaznog neurona koja se dodaje na evaluaciju bazne klase. Izlazna vrijednost može se množiti s bilo kojom vrijednosti, dajući joj veći ili manju moć kontrole nad igrom umjetne inteligencije. Tako na primjer, množenjem procjene s 0.5 nasumično generirane neuronske

mreže dobiva se umjetna inteligencija koja igra otprilike jednako dobro kao i osnovna (u prosjeku blago lošije), ali ovisno o generiranoj mreži igra različita otvaranja (slijed početnih poteza partije). Ne rješava, međutim, problem davanja mata s malim brojem figura.

4.5. Implementacije genetskog algoritma

U sklopu rada implementirana su dva genetska algoritma, jedan uz svaku umjetnu inteligenciju koja koristi neuronsku mrežu. Ovdje će biti detaljnije objašnjena implementacija vezana uz `FNetEvalMinMax`.

Genetski algoritam implementiran je kroz klase `FLeague`, `FPopulation` i funkciju void `GenerateDna`(...). Sustav se koristi na sljedeći način: kreira se instanca klase `FLeague`, na njoj se pozove funkcija `Initialize`(...) te se zatim na njoj ponavljajući zove `Iterate`(). Pozivom funkcije `Initialize` kreira se niz populacija DNA te se postavljaju ograničenja neuronske mreže i mutacije. Populacije se pune nasumično generiranim DNA (moguće je postaviti sve pristranosti neurona i snage veza na nulu kroz makro u standardnom header) koje se dobivaju pozivom funkcije `GenerateDna`. Broj populacija i jedinki u svakoj od njih postavlja se predajom argumenata funkciji `Initialize`.

Funkcija `Iterate` (slika 12) izvodi sve zadatke prelaska jedne generaciju na iduću:

1. Priprema niz partija koje je potrebno odigrati. Za svaku DNA bilo koje populacije, pripremaju se dvije partije s prvom DNA iz svake druge populacije. Ovime su zadovoljena iduća svojstva natjecateljske evolucije:
    a. Za svaki par unutar evolucije, jedinke s kojima se natječu su iste
    b. Zahtjevi sustava nisu fiksni – povisuju se s napretkom populacija
    c. Osigurava se raznolikost – DNA ne može preći iz jedne populacije u drugu čime se dobiva očekivanje različitih DNA u različitim populacijama – što obeshrabruje strogu specijalizaciju neke DNA da bude dobra isključivo protiv neke druge, već općenito dobra
    d. Moguće je održavati „rejting“ populacija – snagu jedinke prve DNA u svake populacije

2. Odigravaju se sve partije potrebne za izračun funkcija dobrote jedinki u više dretvi
3. Rezultati partije primjenjuju se za izračun funkcija dobrote kao suma rezultata svih partija (protiv prvih u populaciji) u kojima je umjetna inteligencija sudjelovala prema turnirskim pravilima; 0 za poraz, 0.5 za remi, 1 za pobjedu. Kako bi se mogle nagraditi ili kazniti manje mutacije, implementirane su još manje promjene na osnovne turnirske bodove:
    a. Za pobjedu, oduzima se broj odigranih poteza podijeljen s 1000
    b. Za poraz, dodaje se broj odigranih poteza podijeljen s 1000
    c. Za remi, dodaje se smanjena razlika zbroja vrijednosti preostalih figura, točnije 0.05 * tanh(Razlika)
4. Kreira se nova generacija svake populacije.
    a. Primjenjuje se funkcija preslikavanja vjerojatnosti propagacije na funkcije dobrote svih jedinki u populaciji (slika 13)
    b. Nasumično se odabire jedinka te se njezina mutirana DNA ubacuje u sljedeću generaciju te populacije

    Moguće je osigurati zadržavanje nepromijenjenu DNA najbolje jedinke postavljanjem makroa `USE_BEST_PRESERVATION` na 1.

```
void FLeague::Iterate()
{
    const int PopCount = Populations.Count();
    TArray<FThreadData> Games;
    for (int LeftPop = 0; LeftPop < PopCount; ++LeftPop)
    {
        for (int LeftUnit = 0; LeftUnit < PopSize; ++LeftUnit)
        {
            for (int RightPop = (LeftUnit == 0) ? LeftPop + 1 : 0; RightPop < PopCount; ++RightPop)
            {
1               if (RightPop == LeftPop)
                {
                    continue;
                }

                SetupGame(Games, LeftPop, LeftUnit, RightPop, 0);
                SetupGame(Games, RightPop, 0, LeftPop, LeftUnit);
            }
        }
    }

    for (int Index = 0; Index < Games.Count(); ++Index)
    {
2       ChessThreads::QueueTaskBlocking(0, ExecPlay, (void*)&Games[Index]);
    }
    ChessThreads::WaitForAllTasks(0);

    for (int Index = 0; Index < Games.Count(); ++Index)
    {
        FThreadData& Game = Games[Index];
        if (Game.UnitIndexWhite == 0 && Game.UnitIndexBlack == 0)
        {
            RateGame(Game.Board, Game.PopIndexWhite, Game.PopIndexBlack);
        }
        const float BoardScore = GameScore(Game.Board);
3       if (Game.UnitIndexWhite == 0)
        {
            Populations[Game.PopIndexBlack].GradeMatch(Game.UnitIndexBlack, BoardScore, Game.MoveCount);
        }
        if (Game.UnitIndexBlack == 0)
        {
            Populations[Game.PopIndexWhite].GradeMatch(Game.UnitIndexWhite, BoardScore, Game.MoveCount);
        }
    }

    for (int Index = 0; Index < PopCount; ++Index)
    {
4       Populations[Index].NextGeneration(*this);
    }
}
```

Slika 12. Programski kod funkcije FLeague::Iterate

```
static float FitnessMapping(float MappedFitness, float WorstFitnesss, float BestFitness)
{
    if (BestFitness == WorstFitnesss)
    {
        return 1.0f;
    }

    const float MinFitness = 0.1f;
    const float Strictness = 2.0f;

    const float Ratio = (MappedFitness - WorstFitnesss) / (BestFitness - WorstFitnesss);
    return LerpF(MinFitness, 1.0f, pow(Ratio, Strictness));
}
```

Slika 13. Programski kod funkcije preslikavanja funkcije dobrote

5. Rezultati

Izvedena je simulacija igranja umjetnih inteligencija, mutacije i evolucije klasom `FLeague`. Korišteno je 10 populacija od 6 jedinki, s uključenim svojstvima zadržavanja najbolje umjetne inteligencije, početka s neutralnim neuronskim mrežama i nasumičnim funkcijama neurona. Dopušteno je do 40 srednjih neurona i 10 rekurentnih parova. Također, svakih 50 generacija onemogućene su daljnje promjene jednom populaciji, redom od prve prema zadnjoj, do zadnjeg isključenja nakon 450 generacija. Već u ranim generacijama vide se promjene koje donose prednost nad baznom umjetnom inteligencijom, kao što se vidi na slici 14.

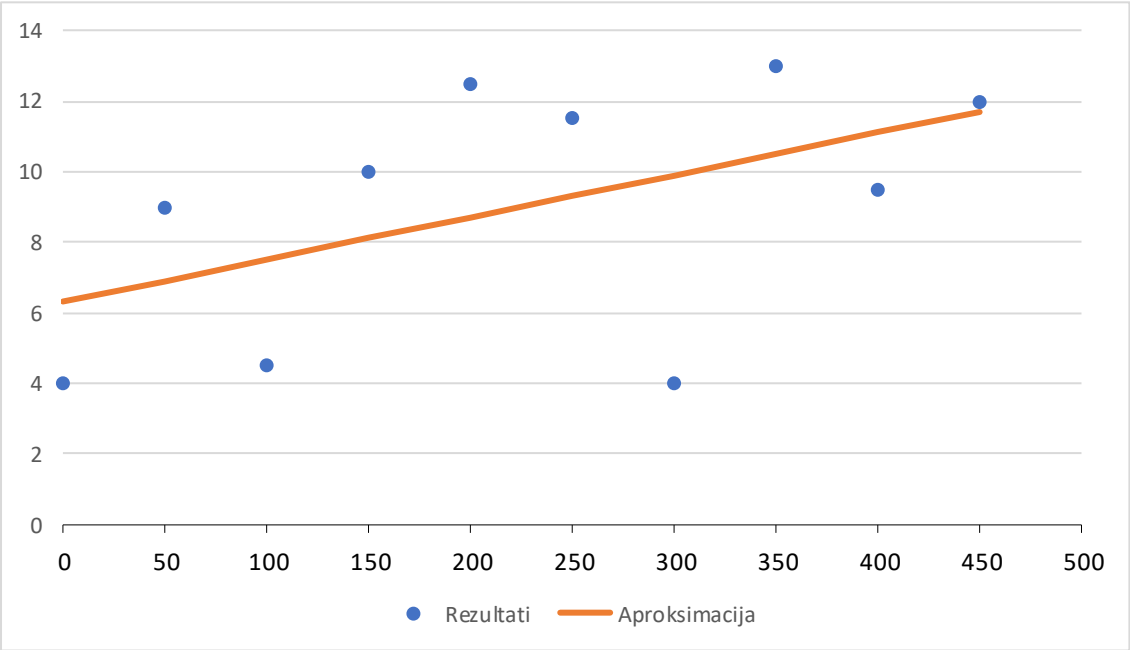
```
[Gen   1] Scores :  9.0  9.0  9.0  9.0  9.0  9.0  9.0  9.0  9.0  9.0
[Gen   2] Scores :  9.0  9.0  9.0  9.0  9.0  9.0  9.0  9.0  9.0  9.0
[Gen   3] Scores :  7.0  7.0  7.0 11.0  7.0 12.0  7.0 12.5  7.0 12.5
[Gen   4] Scores :  6.0  9.0 12.0  7.5 12.0 10.0  6.0 11.5  6.0 10.0
[Gen   5] Scores :  6.0  9.0 12.0  7.5 12.0 10.0  6.5 11.0  6.0 10.0
[Gen   6] Scores :  6.0  9.5 10.5  7.5 12.5  9.0  6.5 12.5  6.5  9.5
[Gen   7] Scores :  6.0 10.0 10.5  7.5 12.5  9.0  6.5 13.0  6.5  8.5
[Gen   8] Scores :  6.0 10.0 10.0  7.5 12.0  9.5  6.5 13.5  6.5  8.5
[Gen   9] Scores :  6.0  9.5  9.5  7.5 12.5 10.5  6.5 14.0  6.5  7.5
[Gen  10] Scores :  6.0  9.5  9.5  7.5 12.5 10.5  6.5 14.0  6.5  7.5
[Gen  11] Scores :  6.0  9.5  9.5  7.5 12.5 10.5  6.5 14.0  6.5  7.5
[Gen  12] Scores :  5.5  9.0  9.5  7.0 12.5 10.0  9.0 14.0  6.0  7.5
[Gen  13] Scores :  5.5  9.0  9.5  7.0 12.5 10.0  9.0 14.0  6.0  7.5
[Gen  14] Scores :  5.5  9.0  9.5  7.0 12.5 10.0  9.0 14.0  6.0  7.5
[Gen  15] Scores :  5.5  9.0  9.5  7.0 12.5 10.0  9.0 14.0  6.0  7.5
[Gen  16] Scores :  5.5  9.0  9.5  7.0 12.5 10.0  9.0 14.0  6.0  7.5
[Gen  17] Scores :  5.5  9.0  9.5  7.0 12.5 10.0  9.0 14.0  6.0  7.5
[Gen  18] Scores :  5.0  8.5 11.5  6.5 13.0 10.0  9.0 14.0  5.0  7.5
[Gen  19] Scores :  5.0  8.5 11.5  6.5 13.0 10.0  9.0 14.0  5.0  7.5
[Gen  20] Scores :  5.0  8.0  9.5  8.0 13.0  9.0  9.0 13.5  7.5  7.5
[Gen  21] Scores :  5.0  8.5 10.5  7.5 13.0  9.0  8.5 13.5  5.0  9.5
[Gen  22] Scores :  5.0  8.5 10.5  7.5 13.0  9.0  8.5 13.0  5.5  9.5
[Gen  23] Scores :  5.0  8.0 10.0  7.5 12.5  9.0  8.5 12.5  7.5  9.5
[Gen  24] Scores :  5.0  8.0 10.0  7.0 12.5  8.5  8.5 12.0  7.5 11.0
[Gen  25] Scores :  5.0  8.0 10.0  7.0 12.5  8.5  8.5 12.0  7.5 11.0
[Gen  26] Scores :  4.0 11.0  8.0  6.0 12.5  8.5  8.5 12.0  7.0 12.5
[Gen  27] Scores :  4.0 11.0  8.0  6.0 12.5  8.5  8.5 12.0  7.0 12.5
[Gen  28] Scores :  4.0 11.0  8.0  5.5 13.0  9.0 10.5 10.5  6.0 12.5
[Gen  29] Scores :  4.0 12.0  9.0  5.0 12.5  9.5 10.0 12.0  4.0 12.0
[Gen  30] Scores :  4.0 12.0  9.0  5.0 12.5  9.5 10.0 12.5  4.5 11.0
```

Slika 14. Prikaz bodova umjetnih inteligencija na početku simulacije.

Kroz 450 generacija dogodio se velik broj evolucija, te su konačni bodovi međusobnog igranja predstavnika (najboljih jedinki) svih populacija sljedeći:

```
4.0  9.0  4.5 10.0 12.5 11.5  4.0 13.0  9.5 12.0
```

Isti rezultati grafički su prikazani na slici 15, aproksimirani linearnom funkcijom



Slika 15. Grafički prikaz rezultata s aproksimacijom

Koeficijent smjera aproksimacije je pozitivan, što uključuje na pozitivnu korelaciju između broja generacije i rezultata. Osim toga, rezultati su razbacani od aproksimacije, što je uzrokovano nasumičnosti mutacija. Povećanjem broj generacija između dva isključenja rezultati bi bili bliže aproksimaciji te bi u nekom trenu počeli formirati monotono rastuću funkciju.

Na tablici 1 vide se rezultati konačnih umjetnih inteligencija svih populacija (osim prve, bazne) protiv bazne umjetne inteligencije.

Tablica 1. Broj bodova (od dva) protiv bazne umjetne inteligencije na raznim dubinama minmax algoritma. Za svaki meč odigrane su dvije partije, gdje svaka umjetna inteligencija jednom igra s bijelim figurama, jednom s crnima.

	50	100	150	200	250	300	350	400	450
1/2	1	1	0.5	0	1	1	2	1	0.5
2/4	2	1.5	1.5	2	1.5	1	1.5	1	1.5
3/6	1	1.5	1	0	0.5	1	1.5	0.5	0.5
4/8	0.5	1	0.5	1	1	1	0	0	1

Treniranje je rađeno na dubinama 2/4, gdje se vide najbolji rezultati. Iste promjene nisu se nužno pokazale korisnim drugim dubinama.

Ova simulacija izvodila se otprilike 5 sati na srednje jakom računalu. Simulacijom s više većih populacija kroz više generacija dobili bi se bolji rezultati. Veća simulacija zahtijevala bi znatno više vremena, i/ili brži procesor s više jezgri.

5.1. Daljnji razvoj

Za daljnji razvoj postoji niz opcija i za minmax algoritam sam po sebi, i za bolji utjecaj neuronske mreže.

Neka od bitnijih poboljšanja za minmax algoritam:

- Pravilna detekcija kraja igre, umjesto provjere za nedostatak kralja. Trenutna implementacija ne razumije da je pat remi nego ga smatra pobjedom.
- Optimizacije alfa-beta rezanja kao npr. sortiranjem poteza tako da se prvo računaju potezi uzimanja, ili pamćenje već evaluiranih pozicija
- Dinamično određivanje dubina ovisno o broju figura na ploči – s manjim brojem figura može računalo stigne simulirati više poteza nego na punoj ploči u istom vremenu

Jedna od ideja za poboljšanje uz pomoć neuronske mreže bi bilo skup različitih umjetnih inteligencija te ih koristiti zajedno, ili selektivno ovisno o broju figura na ploči. Za dobiti specijalizirane neuronske mreže koje bi igrale otvaranje, središnjicu ili završnicu neuronske mreže mogu se trenirati i na različitim početnim pozicijama.

Osim toga, postoji i ideja korištenja neuronske mreže za odbacivanje poteza u ranijim pozivima minmax funkcija, znatno smanjujući broj poziva po dubini.

6. Zaključak

Iz dobivenih rezultata provedenih testiranja možemo zaključiti nekoliko stvari – genetski algoritam imitiranjem prirodne evolucije povlači njen uspjeh – ali isto tako i njenu sporu brzinu. Promjene između generacija su male, tako da bi se dobila velika promjena u kvaliteti DNA potreban je velik broj generacija.

Osim toga, potrebno je imati izrazito veliku neuronsku mrežu kako bi mogli dobiti neuronska mreža koja može rješavati kompleksni zadatak kao što je odlučivanje poteza u šahu – čak i uz pomoć propagacije unatrag, gdje se pokazalo da je nekoliko stotina neurona nedovoljno za nekoliko poteza. Za samostalnu umjetnu inteligenciju koja igra šah vjerojatno bi bilo potrebno nekoliko desetaka tisuća neurona.

Literatura

[1] Recurrent neural network, Wikipedia. Poveznica: https://en.wikipedia.org/wiki/Recurrent_neural_network; pristupljeno 10. lipnja 2019.

[2] Genetic algorithm, Wikipedia. Poveznica: https://en.wikipedia.org/wiki/Genetic_algorithm; pristupljeno 10. lipnja 2019.

[3] Best way to learn to make chess AI?, Reddit. Poveznica: https://www.reddit.com/r/learnprogramming/comments/6hsx4i/best_way_to_learn_to_make_chess_ai/; pristupljeno 10. lipnja 2019.

[4] Chess Programming Wiki. Poveznica: https://www.chessprogramming.org/Main_Page; pristupljeno 10. lipnja 2019.

[5] Sebastian Lague, Coding Adventure: Chess AI, YouTube. Poveznica: https://www.youtube.com/watch?v=U4ogK0MIzqk; pristupljeno 10. lipnja 2019.

[6] Efficient program to calculate e^x, GeeksforGeeks. Poveznica: https://www.geeksforgeeks.org/program-to-efficiently-calculate-ex/; pristupljeno 10. lipnja 2019.

[7] Nathan Reed, Quick And Easy GPU Random Numbers In D3D11. Poveznica: https://www.reedbeta.com/blog/quick-and-easy-gpu-random-numbers-in-d3d11/; pristupljeno 10. lipnja 2019.

[8] Derivacija tanh(x), WolframAlpha. Poveznica: https://www.wolframalpha.com/input/?i=%28%28e%5Ex+-+e%5E-x%29+%2F+%28e%5Ex+%2B+e%5E-x%29%29+-+tanh%28x%29; pristupljeno 10. lipnja 2019.

[9] AlphaStar: Mastering the Real-Time Strategy Game StarCraft II, Deepmind. Poveznica: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii; pristupljeno 10. lipnja 2019.

[10] Coding Standard, Unreal Engine dokumentacija. Poveznica: https://docs.unrealengine.com/4.26/en-US/ProductionPipelines/DevelopmentSetup/CodingStandard/; pristupljeno 10. lipnja 2019.

Razvoj strategije za igranje šaha

Sažetak

Igre su odličan problem za razvoj, učenje i inovaciju umjetne inteligencije. Šah je jedna od prvih poznatijih igri za koju je rađena naprednija umjetna inteligencija, te je napravljena i u sklopu ovog rada. Šah je igra na ploči između dva igrača koji naizmjenično igraju poteze. Umjetna inteligencija za igranje šaha implementirana je kao algoritam minmax, koji simulira sve moguće scenarije odigravanja određenog broja poteza, ocjenjuje dobivene pozicije te odabire najbolji potez pretpostavljajući da će protivnik igrati savršeno. Procjena pozicije implementirana je kao razlika ukupne vrijednosti figura na ploči oba igrača, te njihovih pozicija na ploči. Na tu procjenu dodane su evaluacije neuronske mreže trenirane genetskim algoritmom, poboljšavajući procjenu. Kako bi se osigurale raznolikost i općenitost zahtjeva genetskog algoritma, populacija je podijeljena u više manjih grupa s nemogućnosti prijenosa DNA iz jedne u drugu grupu.

Ključne riječi: šah, umjetna inteligencija, minmax, genetski algoritam, neuronska mreža

Development of a strategy for playing chess

Summary

Games are a great problem for development, study, and innovation in the field of artificial intelligence. Chess is one of the first more well-known games that had an advanced AI be made for, and one was made for this paper. Chess is a board game for two players who alternate making moves. Artificial intelligence for chess was implemented as a minmax algorithm, which simulates all possible scenarios of playing a fixed number of moves, assesses the reached positions and chooses the best move assuming that its opponent is playing perfectly. The assessment is implemented as the difference of values of all the pieces on the board of both players, and their positions on the board. Added to these assessments are neural network's evaluations of the same positions, improving the assessment quality. This network can be trained using the genetic algorithm. To improve variance and generality of demands of the genetic algorithm, its population is spread into several smaller groups, preventing the spread of a single DNA from one group to another.

Key words: chess, artificial intelligence, minmax, genetic algorithm, neural network