Rješavanje problema pametnog agenta za igru šah metodama pretraživanja prostora stanja

Goršić, Leo

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 698

RJEŠAVANJE PROBLEMA PAMETNOG AGENTA ZA IGRU ŠAH METODAMA PRETRAŽIVANJA PROSTORA STANJA

Leo Goršić

Zagreb, veljača 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 698

RJEŠAVANJE PROBLEMA PAMETNOG AGENTA ZA IGRU ŠAH METODAMA PRETRAŽIVANJA PROSTORA STANJA

Leo Goršić

Zagreb, veljača 2025.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 30. rujna 2024.

DIPLOMSKI ZADATAK br. 698

Pristupnik:	**Leo Goršić (0036523443)**
Studij:	Računarstvo
Profil:	Računarska znanost
Mentor:	izv. prof. dr. sc. Marko Đurasević

Zadatak: **Rješavanje problema pametnog agenta za igru šah metodama pretraživanja prostora stanja**

Opis zadatka:

Opisati problem izrade pametnog agenta za igru šah koristeći metode pretraživanja prostora stanja. Istražiti različite prikaze stanja igre šah i način na koji se mogu koristiti u pretraživačkim algoritmima. Osmisliti i testirati različite strategije pretraživanja prostora stanja, poput minimax algoritma s alfa-beta rezanjem, s ciljem poboljšanja performansi agenta u složenim situacijama. Osmisliti načine prilagodbe heurističkih funkcija i njihovu primjenu u svrhu bržeg pronalaženja optimalnih poteza. Prilagoditi metode pretraživanja i ocjenjivanja stanja kako bi agent mogao donositi odluke u realnom vremenu s ograničenim resursima. Radu priložiti izvorne tekstove programa, dobivene rezultate uz potrebna objašnjenja i korištenu literaturu.

Rok za predaju rada: 14. veljače 2025.

Zahvaljujem se svima koji su mi pomogli u izradi ovog rada. Posebno se zahvaljujem mentoru prof. Đuraseviću na strpljenju i pomoći.

Sadržaj

1. Uvod

1.1. Igra šah

Šah je apstraktna društvena strateška igra koja ne uključuje skrivene informacije niti elemente slučajnosti. Igra se na šahovskoj ploči sa 64 polja poredana u mrežu 8×8. Igrači, koji se općenito nazivaju "bijeli" i "crni", svaki kontrolira šesnaest figura: jedan kralj, jedna dama, dva topa, dva lovca, dva skakača i osam pješaka. Bijeli kreće prvi, a zatim crni, te se potezi dalje izmjenjuju. Cilj igre je matirati (zaprijetiti neizbježnim zarobljavanjem) neprijateljskog kralja. Također postoji nekoliko načina na koje igra može završiti neodlučeno.

Zabilježena povijest šaha seže barem do pojave slične igre, chaturanga, u Indiji u sedmom stoljeću. Nakon uvođenja u Perziju, proširio se u arapskom svijetu, a zatim u Europi. Pravila šaha kakva su danas poznata pojavila su se u Europi krajem 15. stoljeća, uz standardizaciju i univerzalno prihvaćanje do kraja 19. stoljeća. Danas je šah jedna od najpopularnijih svjetskih igara, s milijunima igrača diljem svijeta [1].

Veliki dio šahovske teorije razvijen je od samih početaka igre. Počinje se raspravljati o *umjetnosti* u šahovskoj kompoziciji, a šah je zauzvrat utjecao na zapadnu kulturu i umjetnost, te se uočavaju veze s drugim područjima kao što su matematika, računarstvo i psihologija. Jedan od ciljeva ranih računalnih znanstvenika bio je stvoriti stroj za igranje šaha. Godine 1997. Deep Blue postao je prvo računalo koje je pobijedilo aktualnog svjetskog prvaka u meču kada je pobijedio Garryja Kasparova. Današnji šahovski programi znatno su jači od najboljih ljudskih igrača i duboko su utjecali na razvoj šahovske teorije; međutim, šah nije riješena igra.

1.2. Pravila

Za potrebe razumijevanja problematike igre šaha ukratko se navode pravila. Za detaljnije informacije i razumijevanje zainteresirane čitatelje upućujemo na literaturu organizacije FIDE (Fédération Internationale des Échecs; "Internacionalna Šahovska Federacija"), koja je krovno tijelo igre šah. U priručniku su navedena sljedeća pravila.

1.2.1. Ploča i figure

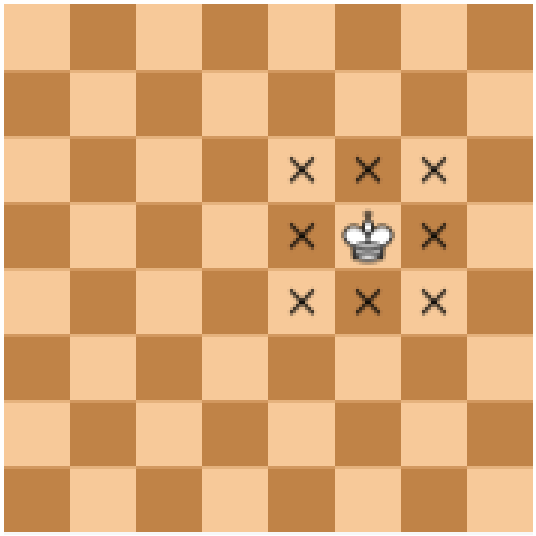
Šahovske figure podijeljene su u dvije grupe, obično svijetle i tamne boje, koje se nazivaju bijele i crne, bez obzira na stvarnu boju ili dizajn. Svaka grupa sastoji se od šesnaest figura: jedan kralj, jedna kraljica, dva topa, dva lovca, dva skakača i osam pješaka. Igra se na kvadratnoj ploči od osam redova i osam stupaca. Iako to ne utječe na igranje igre, prema konvenciji 64 kvadrata mijenjaju se u boji i nazivaju se svijetlim i tamnim kvadratima. Za početak igre, figure se postavljaju na ploču kako je prikazano na slici 1.1.

Slika 1.1. Početna pozicija šahovskih figura

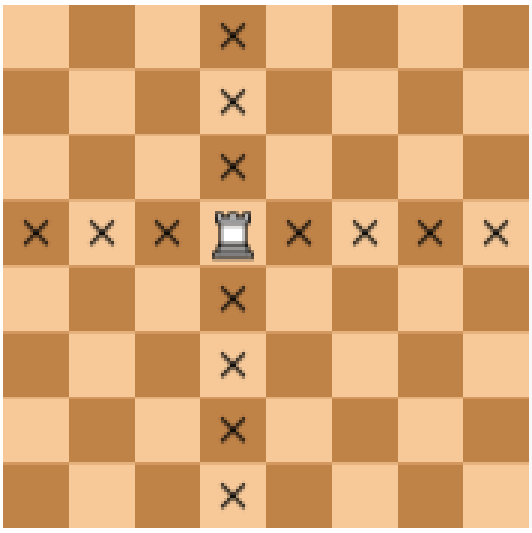
1.2.2. Kretanje figurica

Bijeli prvi kreće, nakon čega se igrači izmjenjuju, pomičući jednu figuru po potezu (osim za rokadu, kada se pomiču dvije figure). Figura se premješta ili na nezauzeto polje ili na polje koje je zauzeto protivnikovom figurom, koja se uzima i uklanja iz igre. Povlačenje poteza je obavezno; igrač ne smije preskočiti potez, čak i kada je potez štetan. Svaka figura ima svoj način kretanja. Na dijagramima, križići označavaju polja na koja se figura može pomaknuti ako nema figura bilo koje boje (osim skakača, koji preskače sve figure između). Sve figure osim pješaka mogu uhvatiti neprijateljsku figuru ako se nalazi na

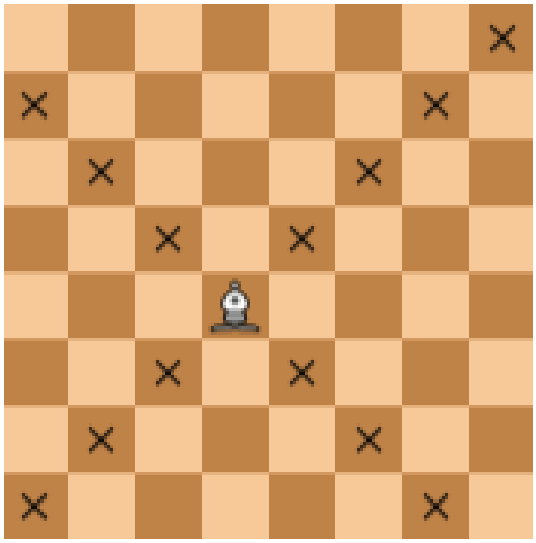
polju na koje se mogu pomaknuti, ako polje nije zauzeto. Figurama općenito nije dopušteno kretanje kroz polja koja zauzimaju figure bilo koje boje, osim za skakača i tijekom rokade.



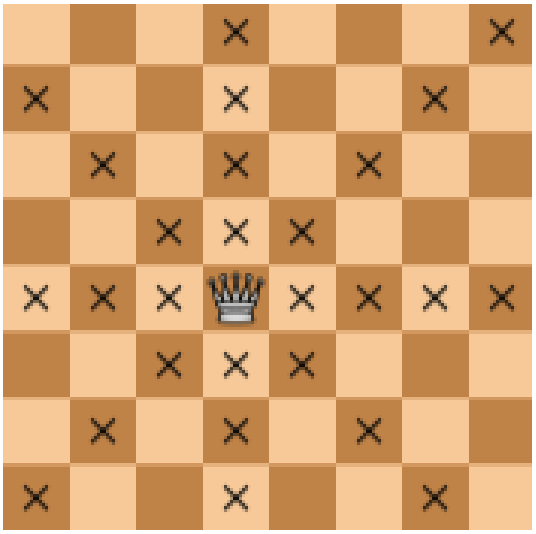
Slika 1.2. Kretanje kralja



Slika 1.3. Kretanje topa



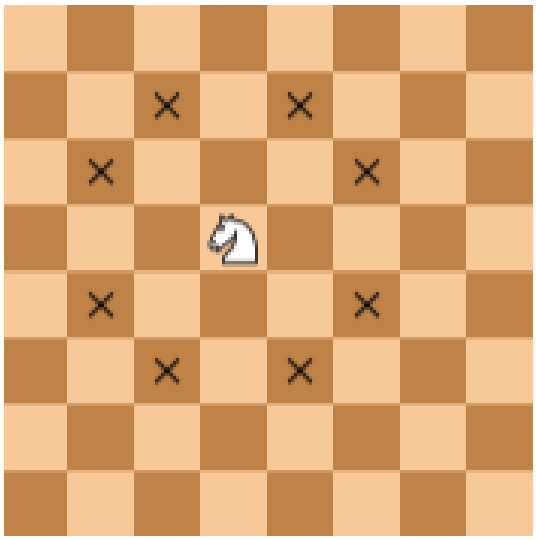
Slika 1.4. Kretanje lovca



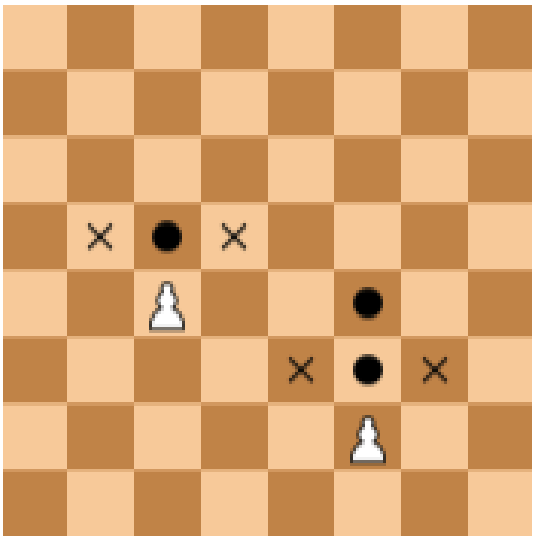
Slika 1.5. Kretanje kraljice

- Kralj se može pomaknuti za jedno polje u bilo kojem smjeru. Postoji i poseban potez koji se zove rokada koji uključuje pomicanje kralja i topa. Kralj je najvrjednija figura—napadi na kralja moraju se odmah ukloniti, a ako je to nemoguće, to se naziva: mat (ili šah-mat) i partija se gubi [2].

- Top se može pomicati za bilo koji broj polja u nizu, ali ne može preskočiti druge figure. Zajedno s kraljem, tijekom kraljeve rokade uključen je i top [2].



Slika 1.6. Kretanje skakača



Slika 1.7. Kretanje pješaka

- Lovac se može pomicati za bilo koji broj polja dijagonalno, ali ne može preskakati druge figure [2].

- Kraljica kombinira moć topa i lovca i može se pomicati za bilo koji broj polja duž reda, niza ili dijagonale, ali ne može preskočiti druge figure [2].

- Skakač se pomiče na bilo koje od najbližih polja koja nisu na istom redu, stupci ili dijagonali. (Stoga potez formira "L"-oblik: dva polja okomito i jedno polje vodoravno, ili dva polja vodoravno i jedno polje okomito.) Skakač je jedina figura koja može preskočiti druge figure [2].

- Pješak se može pomaknuti naprijed do nezauzetog polja neposredno ispred njega na istom stupcu, ili pri svom prvom potezu može pomaknuti dva polja duž istog stupca, pod uvjetom da su oba polja nezauzeta (crne točke na dijagramu). Pješak može uhvatiti protivničku figuru na polju dijagonalno ispred njega pomicanjem na to polje (crni križići). Ne može uhvatiti figuru dok napreduje duž istog stupca, niti se može pomaknuti na bilo koje polje dijagonalno ispred bez hvatanja. Pješak ima dva posebna poteza: en-passant[1] i promocija[2] [2].

[1]En-passant je poseban potez koji se može izvesti samo ako je pješak na početnom položaju i protivnički pješak pomiče se dva polja naprijed, tada se pješak može uhvatiti kao da se pomiče samo jedno polje.

[2]Promocija pješaka je poseban potez koji se može izvesti kada pješak dosegne zadnji red ploče. Pješak se može promijeniti u bilo koju drugu figuru, osim kralja

1.2.3. Šah i mat

Kada je kralj pod neposrednim napadom, kaže se da je pod šahom. Potez kao odgovor na šah je ispravan samo ako rezultira u poziciji u kojoj kralj više nije u šahu.
Postoje tri načina za suzbijanje šaha:

- Pojesti figuricu koja zadaje šah (koja napada kralja).
- Postaviti figuru između figure koja zadaje šah i kralja (što je moguće samo ako je napadačka figura dama, top ili lovac i postoji barem jedno polje između nje i kralja).
- Pomaknuti kralja na polje gdje nije napadnut [2].

Rokada nije dopušteni odgovor na šah [2].
Cilj igre je matirati protivnika; to se događa kada je protivnički kralj u šahu, a ne postoji legalni način da ga se izvuče iz šaha. Nikada nije legalno da igrač napravi potez kojim stavlja ili ostavlja svog kralja pod šahom (napadom) [2].

1.2.4. Pobjeda

Igra se može dobiti na sljedeće načine:

- Šah-mat: protivnički kralj je u šahu, a protivnik nema legalan potez. (Vidi poglavlje šah i mat gore.)
- Predaja: igrač može odustati, prepuštajući igru protivniku [2]. Međutim, ako protivnik nema načina da matira igrača koji je odustao, to je remi prema FIDE pravilima [2]. Većina turnirskih igrača smatra dobrim bontonom podnijeti ostavku u beznadnoj poziciji [3, 4].
- Pobjeda na vrijeme: U igrama s vremenskom kontrolom, igrač pobjeđuje ako protivniku istekne vrijeme, čak i ako protivnik ima nadmoćni položaj, sve dok igrač ima teoretsku mogućnost matirati protivnika u nastavku igre.
- Diskvalifikacija: igrač koji vara, krši pravila ili pravila ponašanja navedena za određeni turnir može biti oduzet [2].

Posljednji način nije relevantan za računalni šah ali je naveden radi potpunosti.

1.2.5. Remi (neodlučena partija)

Postoji nekoliko načina na koje igra može završiti remijem:

- Pat: Ako igrač koji treba krenuti nema legalan potez, ali nije u šahu, pozicija je pat i partija je neodlučena.
- Mrtva pozicija: Ako niti jedan igrač ne može matirati drugoga bilo kojim legalnim nizom poteza, partija je neriješena. Na primjer, ako su samo kraljevi na ploči, a sve ostale figure su osvojene, mat je nemoguć i partija je neriješena prema ovom pravilu.
- Remi po dogovoru: U turnirskom šahu, remi se najčešće postižu međusobnim dogovorom između igrača.
- Trostruko ponavljanje: Ovo se najčešće događa kada niti jedna strana ne može izbjeći ponavljanje poteza bez snošenja štete.
- Pravilo 50 poteza: Ako tijekom prethodnih 50 poteza niti jedan pješak nije pomaknut niti je izvršeno uzimanje (*jedenje*), bilo koji igrač može zahtijevati remi.
- Neriješeno na vrijeme: U igrama s vremenskom kontrolom, partija je neriješena ako je igrač izvan vremena i nijedan niz ispravnih poteza ne bi dopustio protivniku da matira igrača [2].
- Neriješeno odustajanjem: Prema FIDE pravilima, partija je neriješena ako igrač odustane i nijedan niz ispravnih poteza ne bi dopustio protivniku da matira tog igrača [2].

1.3. Šahovska notacija

1.3.1. Algebarska notacija

Povijesno gledano, mnogo različitih sustava notacije korišteno je za bilježenje šahovskih poteza; standardni sustav danas je kratka algebarska notacija. U ovom sustavu svaki je kvadrat jedinstveno identificiran skupom koordinata, a–h za stupce praćene 1–8 za redove. Uobičajeni format je:

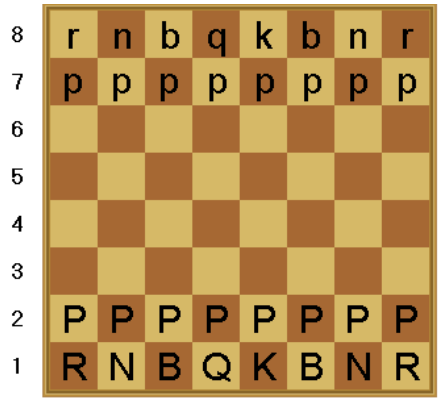
inicijal pomaknute figure – stupac odredišnog polja – redak odredišnog polja

Figurice su identificirane svojim inicijalima. Na engleskom su to K (king), Q (queen), R (rook), B (bishop) i N (knight; N se koristi da bi se izbjegla zabuna s kraljem). Na primjer, Qg5 znači "dama prelazi na g-stupac, 5. red" (to jest, na polje g5).

1.3.2. Forsyth–Edwardsova notacija (FEN)

Forsyth-Edwardsova notacija (FEN) standardna je notacija za opisivanje određene pozicije na ploči u šahovskoj igri. Svrha FEN-a je pružiti sve potrebne informacije za ponovno pokretanje igre s određene pozicije. FEN zapis definira određenu poziciju u igri, sve u jednom retku teksta i koristeći samo ASCII skup znakova.

FEN zapis sadrži šest polja, svako odvojeno razmakom. Polja su sljedeća [5]: Podaci o postavljanju figurica: opisan je svaki redak ploče, počevši od retka 8 do retka 1, s "/" između svakog od njih; unutar svakog retka, sadržaji kvadrata opisani su redom od a-stupca do h-stupca. Svaka figura označena je jednim slovom preuzetim iz standardnih engleskih naziva u algebarskoj notaciji (pješak = "P", skakač = "N", lovac = "B", top = "R", dama = "Q" i kralj = "K"). Bijele figure označene su velikim slovima ("PNBRQK"), dok crne figure koriste mala slova ("pnbrqk"). Skup od jednog ili više uzastopnih praznih polja unutar ranga označava se znamenkom od "1" do "8", što odgovara broju polja.
Tako za početnu poziciju šahovskih figura imamo sljedeći FEN zapis:
`rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR`

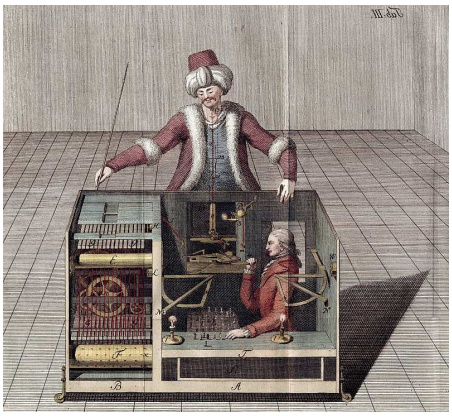


Slika 1.8. Prikaz FEN notacije

1.4. Kratka povijest računalnog šaha

Mehanički Turčin

Najraniji oblik šahovskog stroja pojavljuje se u 18. stoljeću sa strojem nazvanim Mehanički Turčin. Stvorio ga je mađarski izumitelj Wolfgang von Kempelen. To je ljudski model u prirodnoj veličini koji je debitirao 1770. kao prvi autonomni šahovski robot na svijetu. Mehanički Turčin mogao je igrati šah i pobjeđivati protivnike, ići čak i do rješavanja legendarne šahovske zagonetke viteške turneje. Mehanički Turčin ostao je u funkciji od 1770. do 1854. godine, da bi na kraju bio uništen u požaru. Prijevara je otkrivena godinama nakon nestanka stroja, a ljudsko biće je cijelo vrijeme bilo pravi izvor inteligencije Mehaničkog Turčina [6].



Slika 1.9. Prikaz Mehaničkog Turčina u kojem se skrivao čovjek koji je igrao šah. Igrač unutar stroja bio je skriven od pogleda gledatelja. On je imao pogled na šahovsku ploču te bi nitima pokretao lutku koja bi odigrala potez na ploči.

Istraživanja mehaničkog šaha bila su dominantna sve do 1950-ih, kada se pojavilo digitalno računalo. Od tada su šahovski entuzijasti i računalni inženjeri dizajnirali strojeve za igranje šaha i računalne programe sa sve većim stupnjem ozbiljnosti i performansi.

Pojava digitalnog računala i šahovskih programa

Drugi svjetski rat doveo je do rapidnih tehnoloških otkrića, a najveći od njih je izum odnosno stvaranje računala. Dva čovjeka, Alan Turing i Claude Shannon bili su pioniri ovih inovacija, a krajem Drugog svjetskog rata obojica su se zainteresirali za računalni šah. Godine 1950. Claude Shannon objavio je istraživački rad u kojem je detaljno opisao program koji bi potencijalno mogao igrati šah protiv čovjeka. Godinu dana kasnije, Alan

Turing stvorio je prvi računalni algoritam za igranje šaha, no računalna moć u to vrijeme bila je nedovoljna. Turing je ručno testirao svoj algoritam, i iako je sam algoritam bio slab, Turing i Shannon postavili su temelje principe računalnog šaha [7]. Konačno, 1957. IBM-ov inženjer po imenu Alex Bernstein stvorio je prvi potpuno automatizirani šahovski program na svijetu.Program je napravljen za IBM 704 mainframe i bilo mu je potrebno oko osam minuta po potezu [8]. Sposoban za odigravanje cijele partije, ovaj je program označio pravi početak računalnog šaha.

Deep Blue - Prva pobjeda računala nad svjetskim prvakom

Deep Blue je započeo pod imenom ChipTest. ChipTest su razvili i izgradili Feng-hsiung Hsu, Thomas Anantharaman i Murray Campbell u Carnegie Mellonu. Uključili su program u 1986. Sjevernoameričko računalno šahovsko prvenstvo i nisu uspjeli, ali su se sljedeće godine vratili s poboljšanom verzijom i osvojili natjecanje s rezultatom 4-0 [9].

Tim je 1988. godine razvio novi šahovski program pod nazivom Deep Thought. Deep Thought imao je značajne prednosti u odnosu na svoju prethodnu verziju i izdvajao bi se od konkurencije. Postao je prvi program koji je pobijedio velemajstora kada je igrao protiv Benta Larsena u redovnoj turnirskoj igri iste godine kada je izašao [9]. Sljedeće godine Deep Thought osvojio je Svjetsko računalno šahovsko prvenstvo s neporaženim rezultatom 5-0. Šahovski programi još nisu nadmašili ljude, a Deep Thought je izgubio od svjetskog prvaka Garryja Kasparova u dva meča iste godine. Sljedećih godina Deep Thought je ostao prvak među šahovskim programima, da bi na kraju postao Deep Thought 2 i po peti put osvojio Sjevernoameričko računalno šahovsko prvenstvo [9]. IBM je sponzorirao tim počevši od 1994.

Napokon, 1995. IBM-ov tim je objavio novi prototip programa, Deep Blue. Ovaj program je dovršen 1996., a iste se godine prvi put suočio sa šahovskim prvakom Garryjem Kasparovom. Kasparov je pobijedio u meču od šest partija rezultatom 4–2 [9]., ali je to bio prvi put da je šahovski program pobijedio protiv aktualnog šahovskog prvaka. Deep Blue su nadogradili i na njemu su radili i inženjeri i šahovski velemajstori, a godinu dana kasnije tim iz IBM-a dobio je novu priliku. U igri koji će postati ikona, Deep Blue je postao prvi šahovski program koji je pobijedio trenutnog šahovskog prvaka u standardnoj

šahovskoj igri [9]. Unatoč kontroverznim tvrdnjama u Kasparovljevo ime da je IBM varao, rezultat je označio značajno postignuće u šahovskom računarstvu.

Revolucija umjetnih neuronskih mreža

Krajem 2017. inženjeri DeepMinda objavili su šahovski program koji je iznenadio svijet. AlphaZero se temeljio na drugačijem pristupu šahovskom računalstvu, nečemu što do sada nije korišteno. Dok su se prijašnji programi oslanjali na pretraživanje stabala igre i procjenu položaja, AlphaZero se za svoju analizu oslanjao na duboku neuronsku mrežu [10]. To je u biti značilo da AlphaZero može naučiti šah igrajući protiv samoga sebe. Početni testovi s AlphaZero bili su zapanjujući; u meču od 100 igara protiv trenutno najjačeg programa Stockfish, AlphaZero je osvojio 28 igara i izjednačio preostalih 72 [10]. Na mnogo načina AlphaZero je poslužio ne samo kao revolucija za šahovsko računalstvo, već i za svijet umjetne inteligencije općenito. Od 2017. godine prisutnost neuronskih mreža u poznatim šahovskim programima je rasla. Svi poznati šahovski programi današnjice, Leela Chess Zero, Stockfish i Komodo, su uključili neuronske mreže u svoje programe.

2. Temeljne pretpostavke računalnog šaha

2.1. Shannonov broj - broj mogućih partija šaha

Shannon je u svom radu iz 1950. "Programiranje računala za igranje šaha" prikazao izračun za donju granicu složenosti šahovskog stabla igre, što je rezultiralo s oko 10^{120} mogućih partija šaha. Ovaj izračun prikazao je nepraktičnost pretraživanja tzv. *metodom grube sile* [11] Kako bi ilustrirali koliko je to velik broj, često se uspoređuje broj mogućih partija šaha s brojem atoma u poznatom svemiru, koji se procjenjuje na oko 10^{80}.

2.2. Šah je *Zero-sum* igra

Igra s nultim zbrojem (*Zero-sum game*) je matematički prikaz u teoriji igara koja uključuje dva konkurentska igrača, gdje je rezultat prednosti za jednu stranu jednak gubitku za drugu. Drugim riječima, dobitak igrača jedan jednak je gubitku igrača dva, s rezultatom da je njihov zbroj dobitaka jednak nuli. U ovom smislu gubitak se matematički izražava kao dobitak s negativnom prednosti [12]. U gore navedenom smislu, šah je igra s nultim zbrojem. Ako jedan igrač ima prednost, drugi ima isti takav gubitak (odnosno dobitak s negativnim predznakom), a njihov zbroj dobitaka je uvijek nula. Ova činjenica omogućuje računalnim programima da koriste algoritme za optimizaciju koji se temelje na teoriji igara, npr. algoritam Minimax.

2.3. Šah je deterministička igra

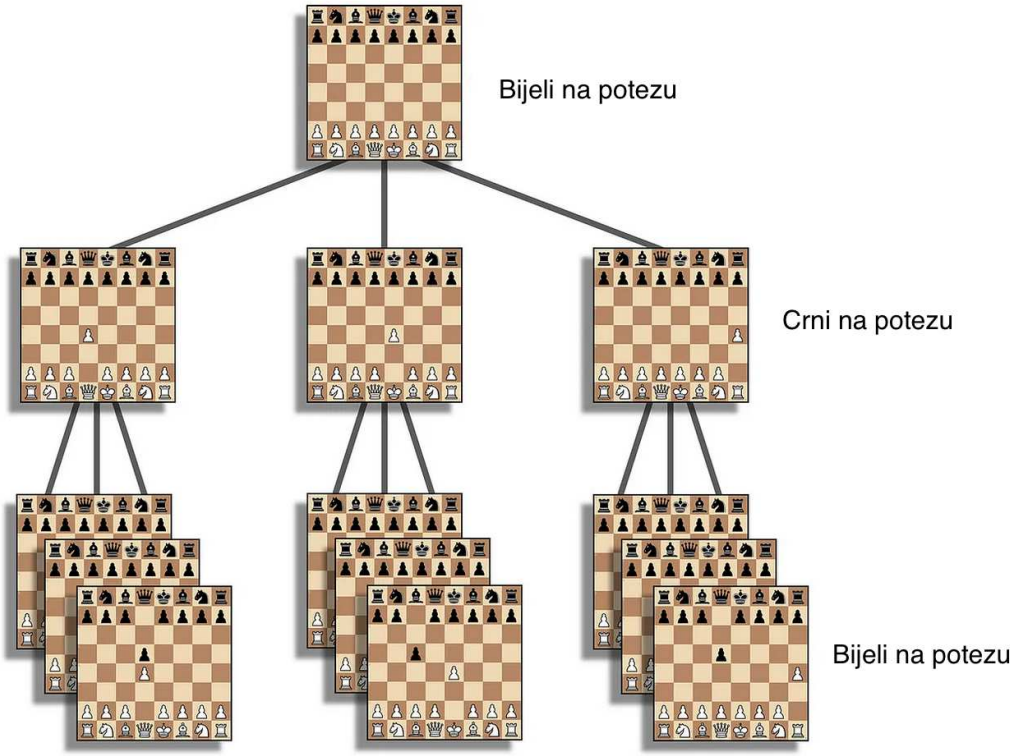
Deterministička igra je ona u kojoj se rezultat igre određuje isključivo na temelju poteza igrača, a ne na temelju slučajnih događaja. U šahu, svaki potez igrača ima određene posljedice, a konačni rezultat igre ovisi isključivo o potezima igrača. Ova činjenica omo-

gućuje računalnim programima da koriste algoritme za pretraživanje stabla igre te da pri tome imaju potpunu informaciju o trenutnom stanju igre. Primjer igara koje nisu determinističke su igre s kartama, kao što je poker, gdje igrači nemaju potpunu informaciju o kartama protivnika.

3. Stablo igre i metode pretraživanja

3.1. Stablo igre

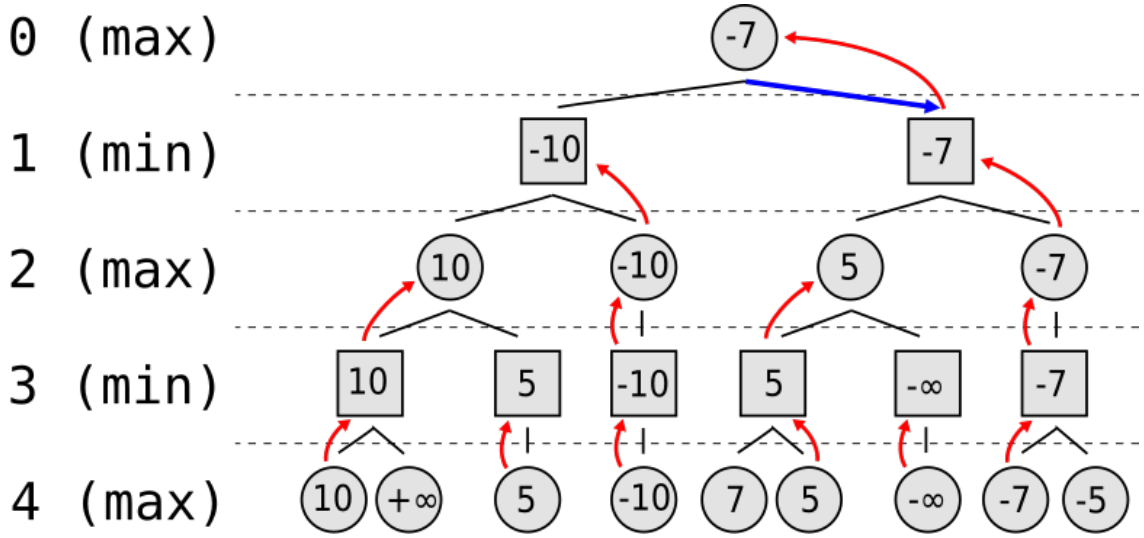
Stablo igre je graf koji predstavlja sva moguća stanja igre unutar deterministčkih igara. Stablo igre može se koristiti za mjerenje složenosti igre, budući da predstavlja sve moguće partije određene igre. Zbog velikih stabala igara složenih igara kao što je šah, koristit će se djelomična stabla igara, što čini izračun izvedivim na računalima. Na slici 3.1. navodi se primjer stabla igre za igru šaha.



Slika 3.1. Prikaz početka stabla igre za igru šaha

3.2. Algoritam minimax

Minimax je pravilo odlučivanja koje se koristi za minimiziranje mogućeg gubitka za scenarij najgoreg slučaja (igrač min), odnosno za maksimiziranje mogućeg dobitka (igrač max). Pretpostavimo da igra koja se igra ima najviše dva moguća poteza po igraču u svakom potezu [1]. Algoritam generira stablo s desne strane, gdje kružići predstavljaju poteze igrača koji izvodi algoritam (maksimizirajući igrač), a kvadratići predstavljaju poteze protivnika (minimizirajući igrač). Zbog ograničenja računalnih resursa, kao što je gore objašnjeno, stablo je ograničeno na pogled unaprijed od 4 poteza. Primjer stabla igre prikazan je na slici 3.2.



Slika 3.2. Prikaz algoritma minimax

Algoritam procjenjuje svaki listni čvor pomoću heurističke funkcije procjene, dobivajući prikazane vrijednosti. Potezi u kojima pobjeđuje maksimizirajući igrač dodijeljena je pozitivna beskonačnost, dok se potezima koji dovode do pobjede minimizirajućeg igrača dodjeljuje negativna beskonačnost. Na razini 3, algoritam će odabrati, za svaki čvor, najmanju vrijednost podređenog čvora i dodijeliti je tom istom čvoru (npr. čvor s lijeve strane će odabrati minimum između "10" i "+beskonačnost", stoga pridjeljujući sebi vrijednost "10"). Sljedeći korak, u razini 2, sastoji se od odabira najveće vrijednosti čvora djeteta za svaki čvor. Još jednom, vrijednosti se dodjeljuju svakom nadređenom čvoru. Algoritam nastavlja naizmjenično procjenjivati maksimalne i minimalne vrijednosti čvorova djeteta dok ne dođe do korijenskog čvora, gdje odabire potez s najvećom vrijednošću

[1] Za usporedbu, Shannon je procijenio za u prosjeku par poteza crnog i bijelog igrača ima 1000 mogućnosti

(predstavljen na slici plavom strelicom). Ovo je potez koji igrač treba napraviti kako bi minimizirao najveći mogući gubitak [13].

Pokazalo se da u zero-sum igrama poput igre šah postoji elegantniji način formuliranja algoritma minimax. Takav algoritam opisan je u sljedećem odjeljku.

3.3. Algoritam negamax

Negamax pretraga je varijanta minimax pretrage koja se oslanja na svojstvo nulte sume vrijednosti dobitka za dva igrača.
Ovaj algoritam se oslanja na činjenicu da je:

$$\min(a, b) = -\max(-b, -a)$$

kako bi se pojednostavila implementacija minimax algoritma. Točnije, vrijednost pozicije za igrača A u takvoj igri je negacija vrijednosti pozicije za igrača B. Dakle, igrač na potezu traži potez koji maksimizira negaciju vrijednosti koja proizlazi iz poteza. Ovakav pristup omogućuje jednostavniju implementaciju algoritma, jer se umjesto dva različita algoritma koristi samo jedan koji poopćuje odlučivanje te ga jednakog primjenjuju i igrač *min* i igrač *max*.

Većina današnjih programa za igranje šaha koristi upravo negamax, jer je jednostavniji za implementaciju i jednako učinkovit kao i klasični minimax algoritam.

3.4. Alfa-beta podrezivanje

Alfa-beta podrezivanje je algoritam pretraživanja koji nastoji smanjiti broj čvorova koje procjenjuje minimax algoritam u njegovom stablu pretraživanja [14]. Algoritam održava dvije vrijednosti, alfa i beta, koje pojedinačno predstavljaju minimalni rezultat koji je osiguran igraču koji maksimizira i maksimalni rezultat koji je osiguran igraču koji smanjuje. U početku, alfa je negativna beskonačnost, a beta pozitivna beskonačnost, tj. oba igrača počinju s najgorim mogućim rezultatom. Kad god maksimalni rezultat koji je osiguran igraču koji minimizira (tj. "beta" igrač) postane manji od minimalnog rezultata koji je

osiguran igraču koji maksimizira (tj. "alpha" igrač) (tj. beta < alpha), maksimizirajući igrač ne mora razmatrati daljnje potomke ovog čvora, jer oni nikada neće biti odabrani. Pretpostavimo da igrač igra šah i da je njegov red. Potez "A" će poboljšati poziciju igrača. Igrač nastavlja tražiti poteze kako bi se uvjerio da možda bolji potez nije propušten. Potez "B" također je dobar potez, ali igrač tada shvaća da će to omogućiti protivniku da odigra šah-mat u dva poteza. Stoga se više ne moraju razmatrati drugi ishodi igranja poteza B budući da protivnik može pobijediti. Maksimalni rezultat koji protivnik može iznuditi nakon poteza "B" je negativna beskonačnost: gubitak za igrača. Ovo je manje od minimalne pozicije koja je prethodno pronađena; potez "A" ne rezultira gubitkom u dva poteza.

```
funkcija alphabeta(čvor, dubina, alpha, beta, maximizirajućiIgrač):
  ako dubina == 0 ili čvor je terminirajući onda:
      vrati vrijednost heuristike za čvor
  ako maksimizirajućiIgrač onda
      vrijednost := -beskonačnost
      za svako dijete čvora:
          vrijednost := max(vrijednost, alphabeta(
            dijete,
            dubina - 1,
            alpha, beta,
            FALSE
            ))
          if vrijednost > beta:
              izađi (* beta odrezivanje *)
          alpha := max(alpha, vrijednost)
      vrati vrijednost
  inače
      vrijednost := +beskonačnost
      za svako dijete čvora:
          vrijednost := min(vrijednost, alphabeta(
            dijete,
            dubina - 1,
```

```
      alpha,
      beta,
      TRUE
      ))
    ako vrijednost < alpha:
        izađi (* alpha odrezivanje *)
    beta := min(beta, vrijednost)
vrati vrijednost
```

Slika 3.3. Pseudokod alfa-beta podrezivanja

4. Heuristička funkcija

Heuristička funkcija, također jednostavno nazvana heuristikom, funkcija je koja rangira alternative u algoritmima pretraživanja na svakom koraku grananja na temelju dostupnih informacija kako bi odlučila koju granu slijediti. Na primjer, može približno odrediti točno rješenje [15].

Cilj heuristike je proizvesti rješenje u razumnom vremenskom okviru koje je dovoljno dobro za rješavanje problema. Ovo rješenje možda nije najbolje od svih rješenja ovog problema ili jednostavno može biti približno točno rješenje. Ali još uvijek je vrijedno jer njegovo pronalaženje ne zahtijeva pretjerano dugo vrijeme. Heuristika je temelj cijelog polja umjetne inteligencije i računalne simulacije razmišljanja, jer se može koristiti u situacijama u kojima ne postoje poznati algoritmi [16].
Heuristička funkcija u računalnom šahu koristi se za procjenu vrijednosti pozicije na šahovskoj ploči. Funkcija može uzeti u obzir različite faktore, kao što su vrijednost figura, pozicija figura na ploči, kontrola središnjih polja, kontrola otvorenih linija, itd. Jedan od jednostavnijih primjera heurističke funkcije je zbrajanje vrijednosti figura na ploči. Ova funkcija često je nazvana materijalna funkcija, jer uzima u obzir samo materijalnu vrijednost figura, zanemarujući njihovu poziciju na ploči.

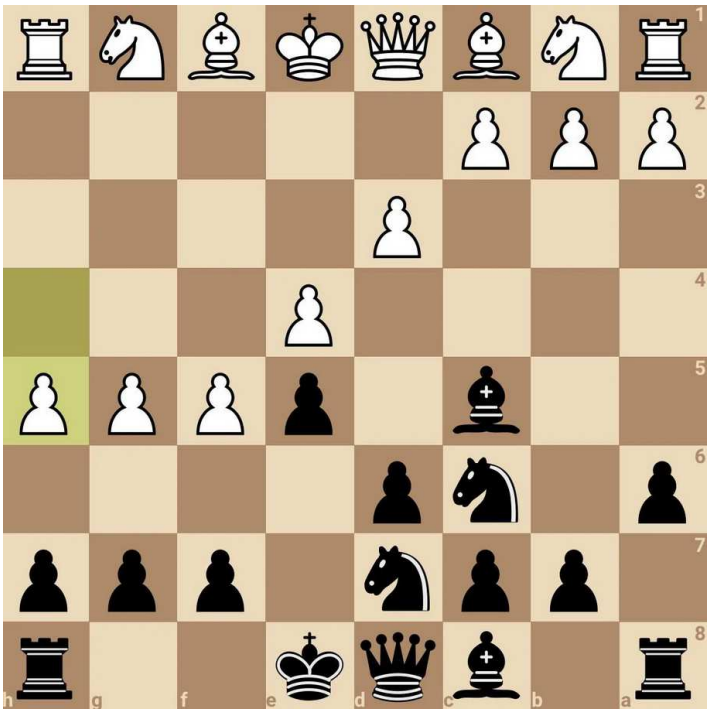
4.1. Materijalna heuristička funkcija

Materijalna heuristička funkcija jednostavno zbraja vrijednosti figura na ploči. Svaka figura ima svoju materijalnu vrijednost, koja se koristi za procjenu ukupne vrijednosti pozicije. Uobičajene materijalne vrijednosti figura u šahu su prikazane na tablici 4.1. Kralj nije naveden, jer obje strane ga nužno imaju za vrijeme trajanja cijele partije. Zbog toga u heurističkoj funkciji figurica kralja se može preskočiti pri zbrajanju. Ova heuristika bi zbrojila sve vrijednosti figurica bijelog igrača. Od te vrijednosti bi zatim oduzela

Figura	Vrijednost (bodovi)
Dama	9
Top	5
Lovac	3
Skakač	3
Pješak	1

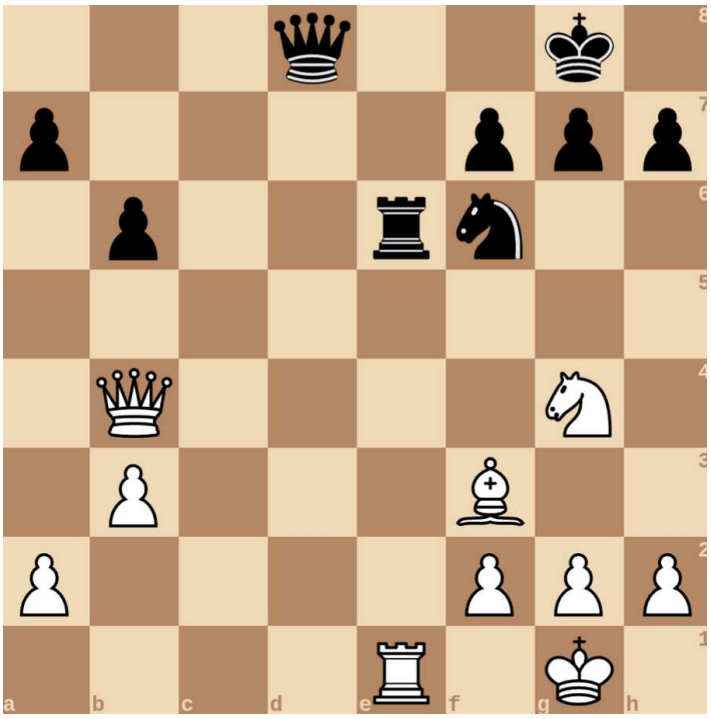
Tablica 4.1. Vrijednosti šahovskih figura

zbroj figurica crnog igrača. Ako je rezultat pozitivan broj, bijeli igrač ima prednost, inače crni igrač ima prednost.



Slika 4.1. Ovu poziciju bi materijalna heuristika procijenila na 0 jer crni i bijeli igrač imaju sve figurice na ploči

Materijalna heuristika ima prednosti vrlo jednostavne implementacije i brze evaluacije. Brza evaluacija je vrlo važna u računalnom šahu, jer se heuristička funkcija mora izračunati za svaku poziciju na ploči tijekom pretraživanja stabla igre. Međutim, materijalna heuristika ima i svoje nedostatke. Zanemaruje poziciju figura na ploči, kao i druge faktore koji mogu biti važni za procjenu vrijednosti pozicije. Na primjer, figura koja kontrolira središnje polje može biti vrijednija od figure koja se nalazi na rubu ploče. Također, figura koja kontrolira otvorenu liniju može biti vrijednija od figure koja je blokirana od strane vlastitih figura. Zbog toga materijalna heuristika može biti neprecizna u mnogim situacijama, što može rezultirati lošim potezima u igri.



Slika 4.2. Ovu poziciju bi materijalna heuristika procijenila na 3 jer je zbroj vrijednosti bijelih figurica 25, a crnih 22

Postoje mnoge heuristike koje se mogu koristiti u računalnom šahu, a materijalna heuristika samo je jedan od primjera. U sljedećem odjeljku opisat će se heuristika koja uzima u obzir poziciju figura na ploči, kao i druge faktore koji mogu biti važni za procjenu vrijednosti pozicije.

4.2. Pojednostavljena funkcija evaluacije - *Simplified evaluation function*

Pojednostavljena funkcija evaluacije je heuristička funkcija koju je originalno formulirao Tomasz Michniewski, poljski računalni znanstvenik i šahovski entuzijast. Funkcija je jednostavna za implementaciju, ali pruža mnogo bolje rezultate od materijalne heuristike. Funkcija uzima u obzir poziciju figura na ploči, kontrolu središnjih polja, kontrolu otvorenih linija, itd. Funkcija se sastoji od nekoliko dijelova, svaki od njih dodjeljuje bodove za određene aspekte pozicije. Bodovi se zatim zbrajaju kako bi se dobila ukupna vrijednost pozicije. Ova heuristika najprije zbraja vrijednosti materijala figura na ploči, poput gore opisane materijalne heuristike, no koristi drugačije vrijednosti figura.
Vrijednost svih figurica pomnožena je sa 100 kako bi se izbjegla aritmetika s decimalnim brojevima. Lovac je postavljen malo vrijednijim od skakača kako bi se favoriziralo

Figura	Vrijednost (bodovi)
Dama	900
Top	500
Lovac	330
Skakač	320
Pješak	100

Tablica 4.2. Vrijednosti šahovskih figura u pojednostavljenoj funkciji evaluacije

posjedovanje para lovaca. Time se ujedno želi izbjeći zamjena lovca za skakača. Ostale figurice (top i dama) imaju iste vrijednosti kao u tradicionalnoj materijalnoj heuristici (te su pomnožene sa 100). Za procjenu pozicije koristi se tablica vrijednosti figurica. Takva tablica ima 8x8 polja. Svaka figurica ima pridruženu tablicu. Obzirom da šahovska ploča i tablica imaju iste dimenzije, svako polje na ploči može se koristiti kao indeks za pristup vrijednosti u tablici. Kao primjer u tablici 4.3. navodim pozicijske vrijednosti za skakača. Ostale tablice mogu se pronaći u literaturi.

-50	-40	-30	-30	-30	-30	-40	-50
-40	-20	0	0	0	0	-20	-40
-30	0	10	15	15	10	0	-30
-30	5	15	20	20	15	5	-30
-30	0	15	20	20	15	0	-30
-30	5	10	15	15	10	5	-30
-40	-20	0	5	5	0	-20	-40
-50	-40	-30	-30	-30	-30	-40	-50

Tablica 4.3. Pozicijske vrijednosti skakača u pojednostavljenoj funkciji evaluacije

Iz tablice 4.3. je vidljivo da se vrijednost skakača maksimizira kada se nalazi u središtu ploče, a minimizira kada se nalazi na rubu ploče. To proizlazi iz toga što skakač na rubu ploče ima manje mogućnosti za kretanje, te napada znatno manji broj polja. Iskustveno, igrači šaha preferiraju skakača središtu ploče. Upravo ta pristranost može se formulirati pomoću ovakve tablice. Vrijednosti pozicije se zbrajaju i za ostale figurice te se ukupna vrijednost heuristike dobiva zbrajanjem svih vrijednosti. Ova heuristika je jednostavna za implementaciju, ali pruža mnogo bolje rezultate od materijalne heuristike. Također pruža razumno brzu evaluaciju, što je važno u računalnom šahu.

5. Reprezentacija šahovske ploče, praćenje stanja igre

Šahovski program treba interni prikaz ploče za održavanje šahovskih pozicija za svoje pretraživanje, procjenu i igru. Osim modeliranja šahovske ploče s njezinim postavljenim figurama, potrebne su neke dodatne informacije za potpuno određivanje šahovske pozicije, kao što su: trenutni igrač na potezu, prava rokade i ostale parametre što objedinjeno nazivamo *stanje igre*. Metode reprezentacije šahovske ploče mogu se podijeliti u dva pristupa. Prvi je pristup usmjeren na figure (Piece Centric), drugi pristup usmjeren je na polja na šahovskoj ploči (Square centric).

Pristup usmjeren na figure - Piece Centic
Reprezentacija usredotočena na figure drži skupove (set) ili liste svih figura još uvijek na ploči - s pripadajućim informacijama koje polje zauzimaju.

Pristup usmjeren na polja - Square Centric
Kvadratno centrično predstavljanje implementira inverznu asocijaciju - je li kvadrat prazan ili je zauzet određenom figurom? Ovaj pristup je često korisniji jer se u šahu često koristi pristup pretraživanja ploče po kvadratima.

U nastavku razmatramo dva pristupa reprezentaciji šahovske ploče usmjerenom na figure.

5.1. Naivna reprezentacija

U naivnoj implementaciji stanje ploče može se predstaviti kao dvodimenzionalno polje (matrica) veličine 8x8. Svaka figura ima svoju jedinstvenu oznaku, npr. "P" za pješaka, "N" za skakača, "B" za lovca, "R" za topa, "Q" za damu i "K" za kralja. Bijele figure oz-

načene su velikim slovima, a crne malim slovima. Prazna polja označena su praznim mjestom. Ova reprezentacija vrlo je jednostavna za implementaciju, ali ima nekoliko nedostataka. Najveći nedostatak ove metode je njezina neefikasnost. Neefikasnost proizlazi iz sporosti baratanja znakovnim nizovima (stringovima) koji predstavljaju figure. Osim toga iteriranje, umetanje i brisanje elemenata iz ovakvih lista je sporo za kontekst računalnog šaha, gdje se nerjetko pretražuje veliki broj pozicija. Na slici 5.1. prikazan je primjer implementacije naivne reprezentacije šahovske ploče u programskom jeziku Python.

```
class Game_state:
    def __init__(self):
        # Ploča je predstavljena kao 2D niz stringova.
        # Prazna pozicija je označena stringom "--".
        # Redovito, pozicija je označena s dva slova.
        # .Prvo slovo (b ili w) označava boju
        # Drugo slovo je figura (Top, Skakač, Lovac, Dama, Kralj).
        self.board = [
            ["bR", "bN", "bB", "bQ", "bK", "bB", "bN", "bR"],
            ["bp", "bp", "bp", "bp", "bp", "bp", "bp", "bp"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["--", "--", "--", "--", "--", "--", "--", "--"],
            ["wp", "wp", "wp", "wp", "wp", "wp", "wp", "wp"],
            ["wR", "wN", "wB", "wQ", "wK", "wB", "wN", "wR"]
        ]
```

Slika 5.1. Primjer implementacije naivne reprezentacije šahovske ploče u programskom jeziku Python

Dakako jezik Python nije najbolji izbor za programe gdje su ključne performanse, no implementacija u programskom jeziku C također bi imala slične probleme. Zbog toga se u praksi koriste druge metode reprezentacije šahovske ploče, koje su efikasnije i brže. Danas svi poznati šahovski programi koriste bitboard reprezentaciju šahovske ploče.

5.2. Bitboard reprezentacija

Bitboard reprezentacija efikasna je metoda reprezentacije šahovske ploče koja koristi binarne nizove (bitove) za predstavljanje figura na ploči. Svaka figura ima svoj binarni niz, gdje svaki bit predstavlja jedno polje na ploči. Tako će biti potrebno ukupno 12 binarnih nizova za predstavljanje svih figura na ploči (6 nizova za bijele figure i 6 nizova za crne figure). Bitboard reprezentacija koristi činjenicu da je šahovska ploča dimenzija 8x8, to jest da je ukupni broj polja na šahovskoj ploči 64. To znači da svako polje na ploči može biti predstavljeno jednim bitom: 1 ako je polje okupirano figurom te 0 ako je polje slobodno. U programskom jeziku C ovakav niz ćemo pohraniti efikasno kao *unsigned long long* tip podataka, koji ima 64 bita.

Dodatno osim za figure koristiti ćemo tri binarna niza (bitboard-a) za praćenje okupacija. Pri tom, nije nam važno koja točno figura okupira dano polje, dovoljno nam je znati je li to figura bijelog ili crnog igrača. To nam pomaže uštediti korake pri generiranju poteza, jer često (pogotovo kada figura *jede* protivničku figuru) nije bitno koja točno figura okupira polje, već samo da je polje okupirano. Na slici 5.2. prikazan je primjer implementacije bitboard reprezentacije šahovske ploče u programskom jeziku C.

```
// definicija bitboard tipa podatka
#define U64 unsigned long long

// bitboard za figure
U64 bitboards[12];

// bitboard za okupacije
U64 occupancies[3];


/*               PRIKAZ POČETNOG STANJA ŠAHOVSKE PLOČE
                              BIJELE FIGURE
      Pješaci                 Skakači              Lovci

8  0 0 0 0 0 0 0 0    8  0 0 0 0 0 0 0 0    8  0 0 0 0 0 0 0 0
7  0 0 0 0 0 0 0 0    7  0 0 0 0 0 0 0 0    7  0 0 0 0 0 0 0 0
```

```
6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0
5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0
4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0
3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0
2  1 1 1 1 1 1 1 1    2  0 0 0 0 0 0 0 0    2  0 0 0 0 0 0 0 0
1  0 0 0 0 0 0 0 0    1  0 1 0 0 0 0 1 0    1  0 0 1 0 0 1 0 0

   a b c d e f g h       a b c d e f g h       a b c d e f g h

       Topovi                Kraljice              Kralj

8  0 0 0 0 0 0 0 0    8  0 0 0 0 0 0 0 0    8  0 0 0 0 0 0 0 0
7  0 0 0 0 0 0 0 0    7  0 0 0 0 0 0 0 0    7  0 0 0 0 0 0 0 0
6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0
5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0
4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0
3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0
2  0 0 0 0 0 0 0 0    2  0 0 0 0 0 0 0 0    2  0 0 0 0 0 0 0 0
1  1 0 0 0 0 0 0 1    1  0 0 0 1 0 0 0 0    1  0 0 0 0 1 0 0 0

   a b c d e f g h       a b c d e f g h       a b c d e f g h

                        CRNE FIGURE

       Pješaci               Skakači               Lovci

8  0 0 0 0 0 0 0 0    8  0 1 0 0 0 0 1 0    8  0 0 1 0 0 1 0 0
7  1 1 1 1 1 1 1 1    7  0 0 0 0 0 0 0 0    7  0 0 0 0 0 0 0 0
6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0
5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0
4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0
3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0
2  0 0 0 0 0 0 0 0    2  0 0 0 0 0 0 0 0    2  0 0 0 0 0 0 0 0
```

```
1  0 0 0 0 0 0 0 0    1  0 0 0 0 0 0 0 0    1  0 0 0 0 0 0 0 0

   a b c d e f g h       a b c d e f g h       a b c d e f g h

      Topovi                Kraljice              Kralj

8  1 0 0 0 0 0 0 1    8  0 0 0 1 0 0 0 0    8  0 0 0 0 1 0 0 0
7  0 0 0 0 0 0 0 0    7  0 0 0 0 0 0 0 0    7  0 0 0 0 0 0 0 0
6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0
5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0
4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0
3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0
2  0 0 0 0 0 0 0 0    2  0 0 0 0 0 0 0 0    2  0 0 0 0 0 0 0 0
1  0 0 0 0 0 0 0 0    1  0 0 0 0 0 0 0 0    1  0 0 0 0 0 0 0 0

   a b c d e f g h       a b c d e f g h       a b c d e f g h

                         OKUPACIJA

Okupacija bijelog      Okupacija crnog     Združena okupacija

8  0 0 0 0 0 0 0 0    8  1 1 1 1 1 1 1 1    8  1 1 1 1 1 1 1 1
7  0 0 0 0 0 0 0 0    7  1 1 1 1 1 1 1 1    7  1 1 1 1 1 1 1 1
6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0    6  0 0 0 0 0 0 0 0
5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0    5  0 0 0 0 0 0 0 0
4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0    4  0 0 0 0 0 0 0 0
3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0    3  0 0 0 0 0 0 0 0
2  1 1 1 1 1 1 1 1    2  0 0 0 0 0 0 0 0    2  1 1 1 1 1 1 1 1
1  1 1 1 1 1 1 1 1    1  0 0 0 0 0 0 0 0    1  1 1 1 1 1 1 1 1
*/
```

Slika 5.2. Primjer implementacije bitboard reprezentacije šahovske ploče u programskom jeziku C

6. Generator poteza

Jedna od ključnih funkcionalnosti šahovskog programa je generiranje legalnih poteza za trenutnu poziciju na ploči. Legalni potezi su potezi koji su u skladu s pravilima kretanja figura u šahu[1]. Generiranje legalnih poteza složen je proces koji uključuje provjeru svih mogućih poteza za svaku figuru na ploči. U nastavku ćemo opisati kako se generiraju legalni potezi za svaku figuru u šahu.

6.1. Naivan generator

Naivni generator koristi naivan prikaz šahovske ploče. On iterira kroz polje figura te za svaku figuru poziva odgovarajuću funkciju koja generira sve legalne poteze za tu figuru. Takvi potezi se pohranjuju u kolekciju koja čuva sve legalne poteze za igrača.

```
self.moveFunctions = {
          'p': self.get_pawn_moves,
          'R': self.get_rook_moves,
          'N': self.get_knight_moves,
          'B': self.get_bishop_moves,
          'Q': self.get_queen_moves,
          'K': self.get_king_moves
      }


def get_rook_moves(self, row, column, moves):
  # gore, lijevo, dolje, desno
  directions = ((-1, 0), (0, -1), (1, 0), (0, 1))
  enemy_color = "b" if self.white_to_move else "w"
  for direction in directions:
```

[1]Poseban slučaj je kada je kralj pod napadom - šahom. Tada je legalan potez samo onaj koji otklanja napad na kralja, otklanja šah. Time se skup mogućih legalnih poteza značajno smanjuje

```
for square_number in range(1, 8):
    end_row = row + direction[0] * square_number
    end_column = column + direction[1] * square_number
    # provjeri je li ta pozicija još uvijek na ploči
    if 0 <= end_row < 8 and 0 <= end_column < 8:
        end_piece = self.board[end_row][end_column]
        # ako polje nije okupirano, onda je to legalan potez
        if end_piece == "--":
            moves.append(Move((row, column), (end_row, end_column)))
        # ovaj potez je legalan, jer možemo pojesti protivničku figuru
        elif end_piece[0] == enemy_color:
            moves.append(Move((row, column), (end_row, end_column)))
            # nakon što pojedemo protivničku figuru, ne možemo ići dalje
            break
        else:
            # ovdje je okupirano polje
            # od strane vlastite figure, ne možemo ići dalje
            break
    else:
        break  # Ne pretražujemo pozicije van ploče
```

Slika 6.1. Primjer implementacije naivnog generatora poteza u programskom jeziku Python

Dan je primjer naivnog generatora za figure topova, za ostale figure koristi se sličan pristup. Zainteresirani čitatelji mogu ih pronaći u literaturi.

Pseudolegalni potezi

Međutim, ovakav generator ne generira legalne poteze. To je zbog toga što ne uvažava pravilo šaha, tj. pravilo da kralj ne smije biti napadnut. Takvi potezi nazivaju se pseudolegalni potezi. Zbog toga je potrebno dodatno filtriranje poteza kako bi se uklonili potezi koji dovode kralja u opasnost.

Naivan algoritam za dobivanje legalnih poteza

```
Algoritam za dobivanje legalnih poteza (uzimajući u obzir šah):
```

```
1. Spremi trenutno stanja igre
   (poput en passant prava, prava rokade, trenutni igrač na potezu).
2. Generiraj sve moguće poteze.
3. Dodaj poteze rokade ako je primjenjivo.
4. Za svaki potez na popisu mogućih poteza (iteriraj u obrnutom redoslijedu):
    a. Napravi potez.
    b. Promijeni trenutnog igrača
       (ako igra bijeli, sada gledamo iz perspektive crnog i obrnuto).
    c. Ako potez stavlja kralja u šah, ukloni ga s popisa valjanih poteza.
    d. Vrati potez.
5. Provjeri ima li valjanih poteza:
    a. Ako je u šahu i nema legalnih poteza,
       postavi šah-mat na vrijednost 1 (True).
    b. Inače, postavi remi na vrijednost 1 (True).
6. Vrati spremljeno stanje igre
   (poput en passant prava, prava rokade, trenutni igrač na potezu)
7. Gotov je popis legalnih poteza.
```

Slika 6.2. Pseudokod za dobivanje valjanih poteza uzimajući u obzir šah

Na slici 6.2. prikazan je algoritam naivnog algoritma za dobivanje legalnih poteza. Nedostatak ovakvog pristupa je očit. Generiranje svih poteza je vrlo sporo. Za svaki potez koji provjeravamo, moramo generirati sve poteze protivnika. Složenost algoritma je eksponencijalna te je praktički neupotrebljiv za dubinu pretrage veću od 4. Zbog toga se u praksi koriste efikasniji algoritmi za generiranje legalnih poteza, poput algoritma koji koristi bitboard reprezentaciju šahovske ploče i bitovnu aritmetiku za brzo generiranje pseudolegalnih poteza.

6.2. Primjer performantog generatora

Performantniji generatori koji koriste bitboard reprezentacije su vrlo složeni algoritmi koji koriste bitovnu aritmetiku za brzo generiranje pseudolegalnih poteza. Na slici 6.3 prikazan je dio implementacije generatora poteza za figure top u programskom jeziku C. Program se ugrubo sastoji od funkcija koje generiraju napade. To su bitboard-ovi koji

imaju postavljene jedinice na koordinatama gdje dane figurice napadaju određeno polje. Za generiranje bitboarda napada koristi se tablica relevantnih bitova te maske koje se predračunavaju na temelju trenutne pozicije figurice i njenih pravila kretanja. Tablica relevantnih bitova je tablica veličine 8x8, na svakom njenom polju (koje korespondira s poljem na šahovskoj ploči) upisan je broj polja koje figura napada osim polja a rubu ploče (jer one ne mijenjaju kretanje ili ponašanje figurice). Sam generator poteza iterira kroz bitboard figure, te za svaku figuru poziva odgovarajuću funkciju koja generira sve legalne poteze za tu figuru. Takvi potezi se pohranjuju u kolekciju koja čuva sve legalne poteze za igrača.

```
// Generator napada topa
const int rook_relevant_bits[64] = {
    12, 11, 11, 11, 11, 11, 11, 12,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    11, 10, 10, 10, 10, 10, 10, 11,
    12, 11, 11, 11, 11, 11, 11, 12};

static inline U64 get_rook_attacks(int square, U64 occupancy)
{
    // dohvati napade topa obzirom na okupaciju
    occupancy &= rook_masks[square];
    occupancy >>= 64 - rook_relevant_bits[square];

    // vrati napade topa
    return rook_attacks[square][occupancy];
}

// Ostali generatori napada
```

```
static inline void generate_moves(moves *move_list) {

// generiranje ostalih figurica

// generiraj poteze topa
if ((side == white) ? piece == R : piece == r)
{
    // iteriraj preko bitboard-a za topa
    while (bitboard)
    {
        // inicijaliziraj izvorno polje (polazište)
        source_square = get_ls1b_index(bitboard);

        // inicijaliziraj sve napade topa u ovisnosti o boji i okupaciji
        attacks =
          get_rook_attacks(source_square, occupancies[both]) &
          ((side == white) ? ~occupancies[white] : ~occupancies[black]);

        // iteriraj preko svih napada topa
        while (attacks)
        {
            // inicijaliziraj ciljno polje (destinaciju)
            target_square = get_ls1b_index(attacks);

            // provjera tihog poteza - služi za quiescence search
            if (!get_bit(((side == white) ?
                occupancies[black] : occupancies[white]), target_square))
                add_move(
                  move_list,
                  encode_move(
                    source_square,
```

```
                    target_square,
                    piece, 0, 0, 0, 0));

            else
                // potez uzimanja figure
                add_move(
                  move_list,
                  encode_move(
                    source_square,
                    target_square,
                    piece, 0, 1, 0, 0, 0));

            // ukloni zadnji bit po signifikantnosti iz trenutnog skupa napada
            pop_bit(attacks, target_square);
        }
    }
}

// generiranje ostalih figura

}
```

Slika 6.3. Primjer generatora pseudolegalnih poteza za topa

7. Odabrani problemi pretrage u kontekstu šahovskog programiranja

U ovom poglavlju razmotriti će se nekoliko odabranih problema pretrage koji se javljaju u kontekstu šahovskog programiranja. Naivne i jednostavne implementacije algoritama često ne adresiraju ove probleme, što dovodi do neprecizne evaluacije ili spore pretrage. Takvi simptomi dovode do nepreciznih poteza i slabije igre. Rješenja su često optimizacije i poboljšanja u pretrazi i evaluaciji koja u konačnici dovode do bolje igre.

7.1. *Quiescence* pretraga

7.1.1. Motivacija i problem horizonta

Zbog složenosti pretrage i velikog broja mogućih poteza, stablo igre u igri šah ima preveliku dubinu za moderna (pa čak i kvantna) računala. Zbog toga se koriste djelomična stabla igre koja su odrezana na određenoj dubini. Međutim, bez obzira na veličinu dubine kojom pretražujemo stablo igre, već prvi potez protivnika koji izlazi iz granica naše pretrage može značajno utjecati na kvalitetu našeg poteza. Primjerice već prvi potez protivnika izvan naše pretrage može dovesti do gubitka igračeve važne figure ili do šah-mata. Ovaj problem naziva se efektom horizonta.

7.1.2. Quiescence - pretraga do *tihe* pozicije

Glavna ideja iza quiescence pretrage je izbjegavanje zaustavljanje pretrage na pozicijama koje su "nestabilne" ili "taktički nestabilne". To su pozicije na kojima su mogući trenutni taktički potezi, kao što su uzimanja (jedenja), promaknuća ili šah. Umjesto zaustavljanja pretrage na tim pozicijama, algoritam selektivno proširuje dubinu pretraživanja do stabilne pozicije. Algoritam evaluira tek poziciju koja je stabilna, tj. *tiha* te smije po-

većati dubinu iznad zadane dubine pretrage ako je potrebno. na slici 7.1. naveden je pseudokod quiescence pretrage:

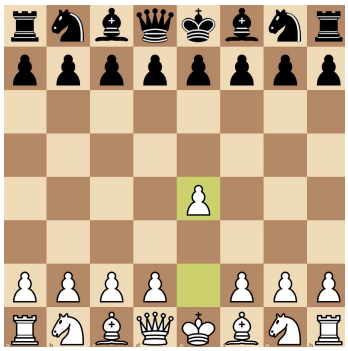
```
  int Quiesce(int alpha, int beta) {
    int evaluacija = Evaluiraj();
    int najbolja_vrijednost = evaluacija;
    if (evaluacija >= beta)
        return evaluacija;
    if (alpha < evaluacija)
        alpha = evaluacija;

    dok( svaki_potez_uzimanja_nije_ispitan ) {
        NapraviUzimanje();
        score = -Quiesce(-beta, -alpha);
        VratiPotez();

        if (score >= beta)
            return score;
        if (score > najbolja_vrijednost)
            najbolja_vrijednost = score;
        if (score > alpha)
            alpha = score;
    }
    return najbolja_vrijednost;
}
```

Slika 7.1. Pseudokod quiescence pretrage. Algoritam nastavlja pretragu sve dok je pozicija nestabilna

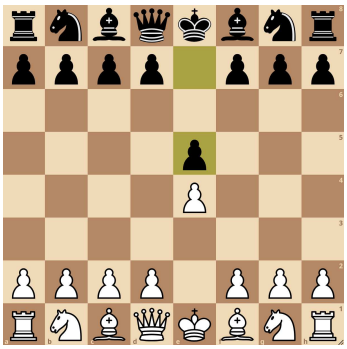
7.2. Transpozicija, *Zobrist hash* funkcija

7.2.1. Motivacija

U šahu često se događa da različiti redoslijedi poteza u konačnici mogu dovesti do iste rezultantne situacije. Pogledajmo primjer na slikama 7.2., 7.3. i 7.4.



Slika 7.2. Prvi potez bijelog je pješak e2e4



Slika 7.3. Zatim crni igra pješaka e7e5

Slika 7.4. Bijeli igra skakača g1f3

Rezultantna situacija je na slici 7.4. na ploči su dva pješaka i skakač. Primjetimo da ćemo do iste rezultantne pozicije doći ako igramo niz poteza kao na slikama 7.5., 7.6. i 7.7.

Slika 7.5. Prvi potez bijelog je skakač g1f3

Slika 7.6. Zatim crni igra pješaka e7e5

Slika 7.7. Bijeli igra pješaka e2e4

Ovaj jednostavan primjer pokazuje kako je moguće imati iste pozicije do kojih se dolazi različitim redoslijedom poteza. Zbog prirode stabla igre, ove dvije iste rezultantne pozicije smatraju se odvojenim pozicijama i svaka od njih se mora evaluirati zasebno. Ovo dovodi do nepotrebnog ponavljanja evaluacije istih pozicija, što značajno usporava pretragu.

7.2.2. Izrada Zobrist hash funkcije

Zobrist hash je konstrukt hash funkcije koja se koristi u računalnim programima koji igraju apstraktne društvene igre, kao što su šah i go. Služi primarno za implementaciju transpozicijskih tablica, posebne vrste hash tablice koja je indeksirana prema poziciji ploče i korištena za izbjegavanje analize iste pozicije više od jednom [17]. Zobristovo raspršivanje nazvano je po svom izumitelju, Albertu Lindseyu Zobristu [18]. Izrada Zobrist hash funkcije počinje nasumičnim generiranjem nizova bitova za svaki mogući element

igre na ploči, tj. za svaku kombinaciju figure i pozicije (u igri šaha to je 6 figura × 2 boje × 64 pozicije) [19]. Sada se bilo koja konfiguracija ploče može razdvojiti na neovisne komponente komada/pozicije, koje se preslikavaju na nasumične bitovne nizove generirane ranije. Konačni Zobrist hash izračunava se kombiniranjem tih nizova bitova korištenjem XOR-a po bitovima. Primjer pseudokoda za Zobrist hash funkciju u igri šah prikazana je na slici 7.8.

```
  konstantni indeksi figura:
    white_pawn := 1
    white_rook := 2
    # etc.
    black_king := 12


func init_zobrist():
    # inicijaliziraj tablicu slučajnim brojevima
    table := a 2-d array of size 64x12
    for i from 1 to 64:  # iteriraj po jednodimenzionalnom polju
        for j from 1 to 12:      # iteriraj po figuricama
            table[i][j] := random_bitstring()
    table.black_to_move = random_bitstring()


func hash(board):
    h := 0
    if is_black_turn(board):
        h := h XOR table.black_to_move
    for i from 1 to 64:      # iteriraj po poljima na ploči
        if board[i] != empty:
            j := the piece at board[i]
            h := h XOR table[i][j]
    return h
```

Slika 7.8. Primjer pseudokoda za Zobrist hash funkciju u igri šah

Pri tome je važno da Zobrist hash funkcija nije kriptografska hash funkcija. Postupak generiranja slučajnih brojeva najčešće je napravljen da bude što jednostavniji i što brži.

Mnogi programi koriste konstantno sjeme *(seed)* za generiranje slučajnih brojeva, što znači da će svaki put kada se program pokrene, generirati iste slučajne brojeve. Ovo je poželjno jer omogućava ponovljivost rezultata i testiranje programa. Međutim, to znači da Zobrist hash funkcija nije otporna na napade i ne može se koristiti za kriptografske svrhe.

7.2.3. Transpozicijska tablica

Šahovski programi analiziraju milijune pozicija koje bi se mogle pojaviti u sljedećih nekoliko poteza igre. Tipično, oni tada ne prate sve do sada analizirane pozicije. U mnogim je igrama moguće doći do određene rezultantne pozicije na više od jednog načina. To se zove transpozicija [20]. U šahu, na primjer, slijed poteza 1. d4 Nf6 2. c4 g6 (vidi algebarsku šahovsku notaciju) ima 4 moguće transpozicije, budući da svaki igrač može zamijeniti svoj redoslijed poteza. Općenito, nakon n poteza, gornja granica mogućih transpozicija je $(n!)^2$. Iako su mnogi od njih nedopušteni nizovi poteza, ipak je vjerojatno da će program završiti analizirajući istu poziciju nekoliko puta.

Kako bi se izbjegao ovaj problem, koriste se transpozicijske tablice. To je tablica s raspršenim adresiranjem *(hash tablica)* svake od dosad analiziranih pozicija do određene dubine. Kada naiđe na novu poziciju, program provjerava tablicu da vidi je li pozicija već analizirana; to se može učiniti brzo, u konstantnom vremenu. Ako tablica sadrži vrijednost koja je prethodno bila dodijeljena ovoj poziciji njena već analizirana vrijednost koristi se izravno. Ako nije, vrijednost se izračunava, a nova pozicija se unosi u hash tablicu.

Broj pozicija koje pretražuje računalo često uvelike premašuje memorijska ograničenja sustava na kojem radi; stoga se ne mogu pohraniti svi položaji. Kada se tablica popuni, manje korištene pozicije se uklanjaju kako bi se napravilo mjesta za nove; to transpozicijsku tablicu čini nekom vrstom predmemorije odnosno *cache* mehanizma [20].

7.3. Poredak poteza pri pretrazi

7.3.1. Motivacija

Poredak poteza pri pretrazi ključan je za brzinu pretrage. Pogrešan poredak poteza može dovesti do nepotrebnog pretraživanja većeg broja pozicija. Poredak poteza važan je jer al-

goritam minimax pretrage sa alfa-beta podrezivanjem pretpostavlja da će najbolji potezi biti prvi generirani. Ako je poredak poteza loš, alfa-beta podrezivanje neće biti učinkovito. Naivno rješenje bi bilo da se generirana lista poteza sortira na način da ih evaluira *glavna* heuristička funkcija (jer ona ocjenjuje vrijednost pojedine pozicije). Međutim to je vrlo neefikasno, jer je poziv heurističke funkcije relativno skup. Cilj bi bio napraviti novu heuristiku koja bi mogla u što kraćem vremenu ocijeniti poteze. Kod takve heuristike naglasak je na što većoj brzini, te se smije žrtvovati i točnost ocjene, budući da će se potezi ionako kasnije evaluirati u dubljim granama pretrage. U idućem odjeljku razmotrit će se strategije za izradu takve heuristike.

Standardna konvencija je da se potezi sortiraju prema sljedećem redoslijedu:

- **PV potezi** - potezi koji su pronađeni u prethodnoj pretrazi
- **Uzimanje figura (MVV/LVA strategija)** - potezi koji uzimaju figure
- **Prvi killer potez** - najučinkovitiji potez bez uzimanja figure na koji se naiđe u određenom sloju tijekom pretrage. Ažurira se čim novi potez izazove beta podrezivanje na toj dubini.
- **Drugi killer potez** - služi kao rezerva i prati sljedeći najbolji potez bez uzimanja figure u istom sloju. Osigurava robusnost ako prvi ubojiti potez nije dovoljno dobar.
- **Ostali potezi** - ostali potezi

7.3.2. PV potezi

PV potezi *(Principal Variation)* su oni koje algoritam pretraživanja smatra najboljim potezima za obje strane za određenu točku u stablu pretraživanja. PV potezi imaju prioritet tijekom redoslijeda poteza jer će najvjerojatnije dovesti do najbolje pozicije. Pretraživanje prvo PV poteza povećava šanse za postizanje rezanja tijekom alfa-beta podrezivanja, što značajno smanjuje broj analiziranih pozicija i skraćuje vrijeme.

Kada neki čvor, odnosno potez ima bolju evaluaciju od bilo kojeg prethodno ispitanog čvora na istoj dubini, on ažurira PV potez. Ovaj je proces rekurzivan: na svakom sloju, najbolji potez iz dublje pretrage ažurira PV te razine.

7.3.3. MVV/LVA strategija

MVV/LVA strategija *(Most Valuable Victim/Least Valuable Aggressor)* je strategija sortiranja poteza koja se temelji na tome da se potezi koji uzimaju figure sortiraju prema

vrijednosti figure koja se uzima i prema vrijednosti figure koja uzima. Ova strategija vrlo je učinkovita jer potezi koji uzimaju figure često dovode do promjene materijalne ravnoteže na ploči. MVV/LVA je jednostavna heuristika za sortiranje poteza uzimanja figura *(jedenja)* razumnim redoslijedom. Unutar takozvanog ciklusa nađi-žrtvu, prvo se traži potencijalna žrtva svih napadnutih protivničkih figura, prema redoslijedu prve najvrjednije, dakle: dama, top, lovac, skakač i pješak. Nakon što se pronađe najvrednija žrtva, ciklus pronalaženja agresora prelazi preko potencijalnih agresora koji bi mogli zarobiti žrtvu obrnutim redoslijedom, od pješaka, skakača, lovca, topa, dame do kralja. Heuristiku je lako implementirati i ona učinkovito pokriva puno jednostavnih slučajeva, kao što je PxR (pješak uzima topa) prije BxP (lovac uzima topa). Međutim, heuristika može biti neuspješna, ako se brane žrtve napadnute od strane vrjednijih napadača, u takvim slučajevima većina programa oslanja se na tablice napada, napade pješaka (obrane) u hodu kako bi izvršili statičku procjenu razmjene [21].

7.3.4. Killer heuristika

Killer heuristika je tehnika sortiranja poteza. Smatra poteze koji su uzrokovali beta podrezivanje u sestrinskom čvoru kao ubojite poteze i sortira ih prioritetno na popisu poteza [22].

7.3.5. Određivanje prioriteta

U redoslijedu poteza, ubojiti potezi obično dolaze odmah nakon PV poteza i (dobrog) uzimanja figure (primjerice MVV/LVA strategija gore opisana). Logika iza ove heuristike je sljedeća: U mnogim pozicijama postoji samo mali skup poteza koji stvaraju prijetnju ili se brane od nje, a oni koji to ne mogu učiniti mogu biti uklonjeni ("ubijeni") istim potezom protivnika [22].
Ubojiti potezi rade na pretpostavci da većina poteza ne mijenja previše situaciju na ploči [22].

7.4. Mjerenje performansi i otklanjanje pogrešaka

7.4.1. Motivacija

Tokom razvoja programa koji igra šah potrebno je u svakom stadiju razvoja mjeriti performanse i otklanjati pogreške. Specifičnost razvoja ovakvih programa je što i male neoptimalnosti mogu dovesti do značajnih usporavanja u stvarnoj igri. Stoga je važno da program bude što efikasniji i da se greške otklanjaju u najranijim fazama razvoja. Brzina rada programa najčešće se mjeri u broju evaluiranih pozicija u sekundi - NPS *(Nodes per second)*. Također može se koristiti i ukupni broj evaluiranih pozicija tijekom pretrage. Za evaluiranje performansi programa koriste se različiti alati i tehnike. U ovom odjeljku razmotrit će se nekoliko alata i tehnika za mjerenje performansi.

7.4.2. *Perft*

Perft, (performance test) je funkcija koja služi za mjerenje performansi komponenti programa za igranje šaha, često se koristi i za otklanjanje pogrešaka. Iterira po stablu igre te generira strogo legalne poteze. Zatim broji sve listne čvorove određene dubine. Oni se zatim mogu usporediti s unaprijed određenim vrijednostima i koristiti za izolaciju grešaka. Perft ignorira neriješene partije ponavljanjem, pravilom od pedeset poteza i nedostatkom materijala. Bilježenjem vremena potrebnog za svaku iteraciju, moguće je usporediti izvedbu različitih generatora pokreta ili istog generatora na različitim programima.

Dolje su navedeni poznati Perft rezultati za početnu poziciju šaha, u zagradi je navedena dubina, a rezultat funkcije je ukupan broj legalnih poteza:

- **Perft(1)** = 20
- **Perft(2)** = 400
- **Perft(3)** = 8,902
- **Perft(4)** = 197,281
- **Perft(5)** = 4,865,609
- **Perft(6)** = 119,060,324

[23]

Na slici 7.9. prikazana je implementacija Perft funkcije u programskom jeziku C.

```
typedef unsigned long long u64;

u64 Perft(int depth)
{
  MOVE move_list[256];
  int n_moves, i;
  u64 nodes = 0;

  if (depth == 0)
    return 1ULL;

  n_moves = GenerateLegalMoves(move_list);
  for (i = 0; i < n_moves; i++) {
    MakeMove(move_list[i]);
    nodes += Perft(depth - 1);
    UndoMove(move_list[i]);
  }
  return nodes;
}
```

Slika 7.9. Prikaz implementacije jednostavnog Perft-a u programskom jeziku C

8. Grafičko korisničko sučelje (GUI)

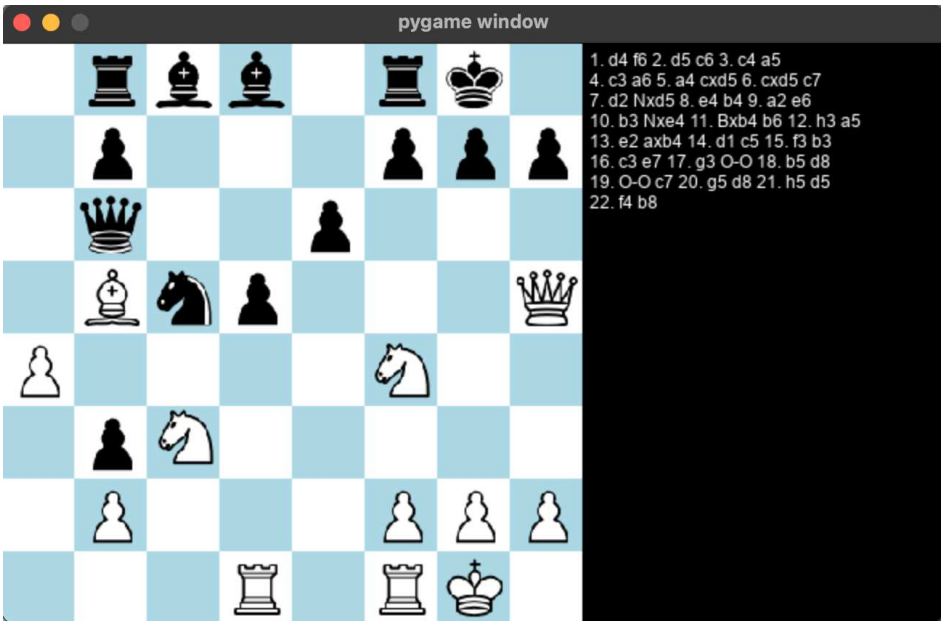
Za potrebe diplomskog rada pored programa za igranje šaha implementirano je jednostavno korisničko sučelje kojim čovjek može igrati šah s računalom. Korisničko sučelje implementirano je u programskom jeziku Python koristeći biblioteku Pygame. Pygame je set besplatnih modula za programiranje igara u programskom jeziku Python. U modulu ChessMain.py učitaju se slike figura (nazvane identično kao i oznake figura), šahovska ploča crta se pomoću pravokutnika gdje zbroj retka i stupca daje parnost polja (bijelo ili crno). Na slici 8.1. prikazan je programski kod za crtanje ploče i figura u programskom jeziku Python.

```
"""
Crtanje polja na ploči.
Gornje lijevo polje je svijetlo (bez obzira na perspektivu!)
"""


def draw_board(screen):
    global board_colors
    board_colors = [game.Color("white"), game.Color("light blue")]

    for row in range(DIMENSION):
        for column in range(DIMENSION):
            color = board_colors[(row + column) % 2]
            game.draw.rect(
              screen, color,
              game.Rect(
                column * SQUARE_SIZE,
                row * SQUARE_SIZE,
```

```
                SQUARE_SIZE,
                SQUARE_SIZE
              )
            )
"""
Crtanje figura na poljima
"""

def draw_pieces(screen, board):
    for row in range(DIMENSION):
        for column in range(DIMENSION):
            piece = board[row][column]

            if piece != "--":
                screen.blit(
                  IMAGES[piece],
                  game.Rect(
                    column * SQUARE_SIZE,
                    row * SQUARE_SIZE,
                    SQUARE_SIZE,
                    SQUARE_SIZE
                  )
                )
```

Slika 8.1. Programski kod za crtanje figura i ploče u programskom jeziku Python

Grafičko korisničko sučelje omogućuje igraču da odabere figuru koju želi pomicati te polje na koje želi pomaknuti figuru. Nakon što igrač odabere potez, program provjerava je li potez legalan. Ako je potez legalan, program pomakne figuru na novo polje. Program koji radi pretragu i odigrava potez napisan je u programskom jeziku C radi performantnosti. Taj se modul pokreće kao zaseban proces te s grafičkim sučeljem komunicira pomoću UCI protokola objašnjenog u nastavku. Slika 8.2. prikazuje izgled implementiranog grafičkog korisničkog sučelja.



Slika 8.2. Grafičko korisničko sučelje

8.1. UCI protokol

UCI (Universal Chess Interface) je protokol koji omogućuje komunikaciju između šahovskog programa i grafičkog korisničkog sučelja. UCI protokol je jednostavan protokol koji se sastoji od niza naredbi koje program šalje grafičkom sučelju. UCI protokol omogućuje brojne funksionalnosti, poput postavljanja pozicije na ploči, traženja najboljeg poteza, postavljanja dubine pretrage, postavljanja vremena pretrage, itd. Za potrebe ovog diplomskog rada koristi se znatno ograničena varijanta UCI protokola koja podržava slijedeće naredbe:

- **position startpos** - naredba koja postavlja početnu poziciju na ploči
- **moves** - naredba koja šalje listu poteza koje je igrač odigrao
- **go depth 10** - naredba koja traži od programa da odigra potez s pretragom određene dubine
- **bestmove** - naredba koju šalje program kako bi obavijestio grafičko korisničko sučelje o najboljem potezu
- **quit** - naredba koja se šalje kako bi se program zatvorio

Osim toga program koji igra šah (chess engine) poziciju učitava i pomoću FEN notacije pomoću position naredbe. Naveden je primjer radi ilustracije:

```
position fen rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq - 0 1
```

9. Rezultati

U sklopu ovog programa implementirani su temelji šahovskog programa. Neke od implementacija preuzete su od drugih autora i prilagođene za potrebe ovog rada. Mnoge implementacije su pojednostavljene i nisu optimalne. U ranijim fazama implementacije najveći nedostatak je bila sporost izvođenja i neefikasnost pretrage. Korištenjem optimizacija opisanih u ovom radu, program je postao znatno brži i efikasniji. U budućem radu moguće je dodati nove optimizacije i tehnike kako bi se program dodatno poboljšao. Također, moguće je dodati nove funkcionalnosti, kao što su podrška za baze podataka otvaranja i završnica, neuronske mreže i strojno učenje.

Implementirani agent je testiran na dubini 10 poteza. Pri tome pretraži okvirno u rasponu od 250000 do 850000 pozicija (ovisno o broju figura i količini legalnih poteza). Na dubinama većim od 10 program se osjetno uspori. Primjera radi, za dubinu 13 na početnoj poziciji programu treba čak 15 puta više vremena da odigra potez i tada pretraži 6539363 pozicije (pri dubini 10 i početnoj poziciji program pretraži 284504 pozicije). Pri dubini 10 program ima rejting od otprilike 1200 ELO[1], što je zadovoljavajući rezultat i agent može pobijediti većinu igrača šaha. U odnosu na prvu verziju implementiranog agenta u programskom jeziku Python program postiže značajno bolje rezultate. Najveći doprinos tome daje bitboard reprezenacija ploče i brže generiranje poteza. Ostali značajni doprinos ubrzanju pretrage je bolja heuristika sortiranja poteza jer se boljim sortiranjem poteza smanjuje broj evaluiranih pozicija. Ostala poboljšanja poput *quiescence* pretrage i transpozicijskih tablica također su doprinijela boljoj igri agenta. Međutim ovaj agent je i dalje značajno sporiji i slabiji od komercijalnih šahovskih programa, kao što je Stockfish[2], koji je i bio inspiracija pri izradi ovog agenta.

[1] Ova pretpostavka se temelji na igrama s igračima poznatog rejtinga

[2] Stockfish je jedan od najboljih šahovskih programa na svijetu i ima rejting od preko 3600 ELO. Stockfish koristi mnoge napredne tehnike i optimizacije koje nisu opisane u ovom radu, kao što su neuronske mreže i strojno učenje, baze podataka otvaranja i završnica, tehnike sortiranja poteza, itd. Stockfish također koristi bitboard reprezentaciju ploče i brze algoritme za generiranje poteza. Stockfish je open-source program i dostupan je na GitHubu.

10. Zaključak

Igra Šah je jedna od najpoznatijih i najpopularnijih igara na svijetu. Šahovski programi su od samih početaka računalne znanosti bili jedan od zahtjevnijih i zanimljivijih problema. Iako je šah igra s pravilima koja su jednostavna, složenost igre dolazi iz velikog broja mogućih poteza i pozicija. Međutim napretci u računalnoj snazi i algoritmima omogućili su razvoj šahovskih programa koji su danas nemogući za pobijediti, čak i od najboljih ljudskih igrača. Ovaj diplomski rad se bavi implementacijom šahovskog programa koji koristi algoritam minimax pretrage s alfa-beta podrezivanjem i ostalim optimizacijama opisanim u ovom radu. Program je implementiran u programskom jeziku C te koristi grafičko korisničko sučelje implementirano u programskom jeziku Python koristeći biblioteku Pygame. U ovom radu opisane su neke od ključnih tehnika i algoritama koji se koriste u šahovskim programima. Opisane su tehnike pretrage, optimizacije i mjerenja performansi.

Međutim, komercijalni šahovski programi koriste mnoge druge tehnike i optimizacije koje nisu opisane u sklopu ovog diplomskog rada. Šahovski programi danas uvelike koriste neuronske mreže i strojno učenje za poboljšanje evaluacije pozicija i poteza. Pored toga, često se koriste i baze podataka otvaranja i završnica kako bi još dodatno poboljšali igru. Najvažniji algoritam je algoritam minimax s alfa-beta podrezivanjem. Ovaj algoritam omogućuje pretraživanje stabla igre na efikasan način. U sklopu ovog rada opisane su i neke od optimizacija ovog algoritma, kao što su quiescence pretraga, transpozicijske tablice, Zobrist hash funkcija, tehnike sortiranja poteza. Heuristička funkcija je ključna za evaluaciju pozicija i poteza. Važno je da ona bude što preciznija ali i brza. U ovom radu opisana je pojednostavljena funkcija evaluacije kao jednostavniji primjer heuristike koja pored materijalne prednosti uzima u obzir i poziciju figura na ploči.

Literatura

[1] J.-L. Cazaux i R. Knowlton, *A World of Chess: Its Development and Variations through Centuries and Civilizations*. McFarland, 2017.

[2] *"Fide Laws of Chess taking effect from 1 January 2023". FIDE. Archived from the original on 1 January 2023. Retrieved 1 January 2023.*, 2023. [Mrežno]. Adresa: https://www.fide.com/fide/handbook.html

[3] D. B. Pritchard, *The Right Way to Play Chess*, 2008. izd. Right Way, 2008.

[4] *"Why Grandmasters Rarely Checkmate". Los Angeles Times. 18 May 2001. Archived from the original on 29 December 2020. Retrieved 3 December 2020.*, 2001.

[5] “"standard: Portable game notation specification and implementation guide". internet archive. 12 march 1994. retrieved 25 july 2020.” 1994. [Mrežno]. Adresa: https://web.archive.org/web/20200312164700/http://www.saremba.de/chessgml/standards/pgn/pgn-complete.htm

[6] S. Schaffer, *"Enlightened Automata". Schaffer, Simon (1999). In Clark et al. (Eds), The Sciences in Enlightened Europe, Chicago and London, The University of Chicago Press, pp. 126–165.*, 1999.

[7] D. Heath, *"The Historical Development of Computer Chess and its Impact on Artificial Intelligence". Heath, David (1997). University of Luton.*, 1997.

[8] B. Wall, *"Chessville – Early Computer Chess Programs – by Bill Wall – Bill Wall's Wonderful World of Chess"*. [Mrežno]. Adresa: https://archive.is/20120721123400/http://www.geocities.com/SiliconValley/Lab/7378/early.htm

[9] *"History of deep blue". IBM. Retrieved 27 February 2016.*, 2016.

[10] D. Silver, J. Schrittwieser, i D. Hassabis, *"AlphaZero: Shedding new light on chess, shogi, and Go". Deepmind. Retrieved 8 April 2022.*, 2022.

[11] C. E. Shannon, *"Programming a computer for playing chess". Shannon, Claude E. (March 1950). Levy, David (ed.).*, 1950.

[12] S. Blakely, *"Zero-Sum Game Meaning: Examples of Zero-Sum Games". Master*

Class., 2022. [Mrežno]. Adresa: https://www.masterclass.com/articles/zero-sum-game-meaning-examples-of-zero-sum-games

[13] [Mrežno]. Adresa: https://en.wikipedia.org/wiki/Minimax

[14] S. Russell i P. Norvig, *"Artificial Intelligence: A Modern Approach". Russell, Stuart; Norvig, Peter (2021).*, 2021.

[15] J. Pearl, *"Heuristics: intelligent search strategies for computer problem solving". Pearl, Judea (1984). United States: Addison-Wesley Pub. Co., Inc., Reading, MA. p. 3. OSTI 5127296.*, 1984.

[16] M. J. Apter, *"The Computer Simulation of Behaviour". Apter, Michael J. (1970). London: Hutchinson Co. p. 83. ISBN 9781351021005.*, 1970.

[17] B. Moreland, *"Zobrist keys: a means of enabling position comparison."*, 1986.

[18] A. L. Zobrist, *"A New Hashing Method with Application for Game Playing". Albert Lindsey Zobrist (1969). Tech. Rep. 88, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin.*, 1969.

[19] [Mrežno]. Adresa: https://www.chessprogramming.org/Zobrist_Hashing#Initialization

[20] F.-D. Laramee, 2002. [Mrežno]. Adresa: https://www.gamedev.net/tutorials/_/technical/artificial-intelligence/transposition-tables-r1175/

[21] [Mrežno]. Adresa: https://www.chessprogramming.org/MVV-LVA

[22] [Mrežno]. Adresa: https://www.chessprogramming.org/Killer_Heuristic

[23] [Mrežno]. Adresa: https://www.chessprogramming.org/Perft_Results

Sažetak

Rješavanje problema pametnog agenta za igru šah metodama pretraživanja prostora stanja

Leo Goršić

Igra Šah je jedna od najpoznatijih i najpopularnijih igara na svijetu. Šahovski programi su od samih početaka računalne znanosti bili jedan od zahtjevnijih i zanimljivijih problema. Iako je šah igra s pravilima koja su jednostavna, složenost igre dolazi iz velikog broja mogućih poteza i pozicija. Međutim napretci u računalnoj snazi i algoritmima omogućili su razvoj šahovskih programa koji su danas nemogući za pobijediti, čak i od najboljih ljudskih igrača. Ovaj diplomski rad se bavi implementacijom šahovskog programa koji koristi algoritam minimax pretrage s alfa-beta podrezivanjem i ostalim optimizacijama opisanim u ovom radu. Program je implementiran u programskom jeziku C te koristi grafičko korisničko sučelje implementirano u programskom jeziku Python koristeći biblioteku Pygame. U ovom radu opisane su neke od ključnih tehnika i algoritama koji se koriste u šahovskim programima. Opisane su tehnike pretrage, optimizacije i mjerenja performansi.

Ključne riječi: Šah; Minimax; Alfa-beta podrezivanje; Quiescence pretraga; Transpozicijske tablice; Zobrist hash funkcija; Pygame

Abstract

Solving the smart agent problem for playing chess using state space search methods

Leo Goršić

The game of Chess is one of the most famous and popular games in the world. Chess programs have been one of the most challenging and interesting problems since the beginning of computer science. Although chess is a game with simple rules, the complexity of the game comes from the large number of possible moves and positions. However, advances in computing power and algorithms have enabled the development of chess programs that are now impossible to defeat, even by the best human players. This thesis deals with the implementation of a chess program that uses the minimax search algorithm with alpha-beta pruning and other optimizations described in this paper. The program is implemented in the C programming language and uses a graphical user interface implemented in the Python programming language using the Pygame library. This paper describes some of the key techniques and algorithms used in chess programs. Search, optimization, and performance measurement techniques are described.

Keywords: Chess; Minimax; Alpha-beta pruning; Quiescence search; Transposition tables; Zobrist hash function; Pygame