ALGORITAM ZA GENERIRANJE ŠAHOVSKOG POTEZA U PROGRAMSKOM JEZIKU C#

Kvesić, David

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA

Sveučilišni studij

ALGORITAM ZA GENERIRANJE ŠAHOVSKOG POTEZA U PROGRAMSKOM JEZIKU C#

Završni rad

David Kvesić

Osijek, 2021.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac Z1P - Obrazac za ocjenu završnog rada na preddiplomskom sveučilišnom studiju

Osijek, 31.08.2021.

Odboru za završne i diplomske ispite

Prijedlog ocjene završnog rada na preddiplomskom sveučilišnom studiju

Ime i prezime studenta:	David Kvesić
Studij, smjer:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4231, 25.07.2018.
OIB studenta:	86621679618
Mentor:	Doc.dr.sc. Tomislav Rudec
Sumentor:	Izv. prof. dr. sc. Alfonzo Baumgartner
Sumentor iz tvrtke:	
Naslov završnog rada:	Algoritam za generiranje šahovskog poteza u programskom jeziku C#
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Predložena ocjena završnog rada:	Izvrstan (5)
Kratko obrazloženje ocjene prema Kriterijima za ocjenjivanje završnih i diplomskih radova:	Primjena znanja stečenih na fakultetu: 3 bod/boda Postignuti rezultati u odnosu na složenost zadatka: 3 bod/boda Jasnoća pismenog izražavanja: 2 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	31.08.2021.
Datum potvrde ocjene Odbora:	08.09.2021.

Potpis mentora za predaju konačne verzije rada u Studentsku službu pri završetku studija:	Potpis:
	Datum:

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA
I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

IZJAVA O ORIGINALNOSTI RADA

Osijek, 10.09.2021.

Ime i prezime studenta:	David Kvesić
Studij:	Preddiplomski sveučilišni studij Računarstvo
Mat. br. studenta, godina upisa:	R4231, 25.07.2018.
Turnitin podudaranje [%]:	9

Ovom izjavom izjavljujem da je rad pod nazivom**: Algoritam za generiranje šahovskog poteza u programskom jeziku C#**

izrađen pod vodstvom mentora Doc.dr.sc. Tomislav Rudec

i sumentora Izv. prof. dr. sc. Alfonzo Baumgartner

moj vlastiti rad i prema mom najboljem znanju ne sadrži prethodno objavljene ili neobjavljene pisane materijale drugih osoba, osim onih koji su izričito priznati navođenjem literature i drugih izvora informacija.
Izjavljujem da je intelektualni sadržaj navedenog rada proizvod mog vlastitog rada, osim u onom dijelu za koji mi je bila potrebna pomoć mentora, sumentora i drugih osoba, a što je izričito navedeno u radu.

Potpis studenta:

SADRŽAJ

1. UVOD

Od samih početaka računalne znanosti zbog velike popularnosti šah je privlačio znanstvenike te računalne inženjere i entuzijaste zbog svoje jednostavnosti igranja, dok je istovremeno bio izrazito kompleksan za usavršiti s obzirom na stratešku složenost igre kao i iznimno velik broj mogućih ishoda. Stoga je izrada računalnog programa koji će samostalno igrati šah i donositi racionalne i logički prihvatljive odluke (*Chess engine*) predstavljala veliki izazov za stručnjake. S vremenom su dolazak novih tehnologija te razvoj znanosti omogućili rapidan razvoj računalnih šahovskih programa koji su počeli parirati, a naposljetku su i nadmašili najbolje svjetske velemajstore. Osnovni je cilj šahovskog programa na temelju situacije na šahovskoj ploči donijeti najbolji mogući potez u prihvatljivom vremenu. Postoje brojne heuristike kojima se dolazi do efikasnih programa, a u ovom radu implementirat će se ona temeljena na *minimax* algoritmu gdje se potezi na šahovskoj ploči pretražuju kao stablo. U drugom poglavlju daje se kratki opis šaha te pravila koja su nužna za implementaciju igre na računalu. U trećem poglavlju navode se tehnologije korištene u implementaciji programskog rješenja. U četvrtom se poglavlju opisuje implementacija osnovnih teoretskih principa na kojima radi šahovski algoritam poput funkcije pretraživanja legalnih poteza i njezine optimizacije te funkcije evaluacije koja procjenjuje stanje na ploči i računa koji je potez najbolji. U petom poglavlju analiziraju se rezultati koji su dobiveni korištenjem programa te su objašnjeni glavni nedostaci programa i moguća rješenja.

1.1. Zadatak završnog rada

Zadatak ovog rada je opisati osnovne principe rada na kojima funkcionira algoritam za generiranje šahovskog poteza te implementirati algoritam u programskom jeziku C#.

2. ŠAH U RAČUNARSTVU

Šah je danas najpoznatija i najpopularnija strateška igra na svijetu. Procjenjuje se da na svijetu ima više od 600 milijuna igrača šaha, a to je dijelom posljedica karakteristika šaha kao igre s jednostavnim pravilima i bez prisustva faktora sreće koju mogu igrati ljudi svih uzrasta neovisno o njihovom društvenom statusu. Vještina igranja šaha postala je sinonim za matematičko logičku inteligenciju. Neki znanstvenici te istraživači prozvali su šah vinskom mušicom umjetne inteligencije [1] aludirajući na stvaranje ranih šahovskih programa koji su iako na prvi pogled jednostavni, postali jedan od temelja ranog proučavanja i razvoja umjetne inteligencije. Von Neumannov[1] *minimax* teorem postavio je temelj za razvoj računalnih šahovskih programa, a Claude Shannon[2] je 1949. godine objavio rad „*Programming a Computer for playing Chess*“ [2] u kojem je opisao strategiju na kojoj bi se zasnivao njegov program za igranje šaha. Program bi za pretraživanje koristio *minimax* algoritam, a za određivanje kvalitete poteza evaluaciju šahovske ploče temeljenu na jačini figura, njihovoj poziciji, mobilnosti figura i drugim faktorima. Sljedeća bitna karakteristika programa je način prikaza stanja na šahovskoj ploči o kojem ovisi i način implementacije samog algoritma. Dvodimenzionalno polje elemenata intuitivan je način prikaza ploče, a koristi se i prikaz s jednodimenzionalnim poljem od 64 elementa kod kojeg je pristup memoriji nešto brži, ali je upravljanje granicama ploče zahtjevnije [3]. Treći popularan način je *bitboard* koncept, gdje svaki bit odgovara jednom polju na ploči. Time se na modernim 64-bitnim arhitekturama cijeli *bitboard* može spremiti u jedan registar [4]. Iako se nije mislilo da će računala dostići ljudsku razinu, daljnji razvoj tehnologija kao i razvoj optimizacijskih tehnika omogućili su neusporedivo bolje performanse računala koja danas razmišljaju desetke poteza unaprijed.

1997. godine superračunalo Deep Blue porazilo je svjetskog prvaka u šahu Garryja Kasparova[3] i taj događaj predstavlja prekretnicu kojom je umjetna inteligencija prestigla ljudsku u igranju šaha. Današnji šahovski programi rade na principima dubokog učenja koristeći neuronske mreže koje na temelju već odigranih partija uče bolje igrati šah i u potpunosti su nadmašili i najbolje svjetske igrače.

[1] John von Neumann, 1903. – 1957., američki matematičar, fizičar i stručnjak za računarsku znanost
[2] Claude Elwood Shannon, 1916. – 2001., američki matematičar, inženjer i kriptograf, otac teorije informacije
[3] Garry Kimovich Kasparov, 1963., ruski šahovski velemajstor, bivši svjetski šahovski prvak

3. O ŠAHU

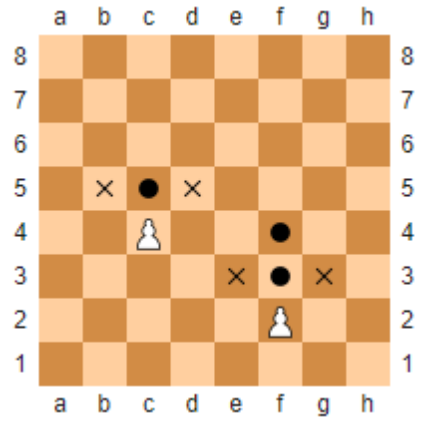
Šah je igra za dva igrača u kojoj uz početni postav figura i pravila njihovog kretanja igrači pomiču po jednu figuru naizmjenično s ciljem postavljanja figure na mjesto protivničkog kralja. Igra se odvija na kvadratnoj ploči s 8×8 polja koja su naizmjenično obojena svijetlom (bijelom) i tamnom (crnom) bojom. Oba igrača igru započinju sa 16 figura. To su osam pješaka, dva skakača, dva lovca, dva topa te kraljica i kralj. Jedan igrač igra sa svijetlim, a drugi s tamnim figurama.

3.1. Početni postav i pravila kretanja figura

Slika 3.1. Početni položaj figura na ploči

Na slici 2.1. prikazan je inicijalni položaj svih figura na šahovskoj ploči. U redovima 2 i 7 nalazi se osam pješaka a u redovima 1 odnosno 8 nalaze se, s lijeva na desno, top, skakač, lovac, kraljica, kralj, lovac, skakač i top. Svaka od figura kreće se na svoj način i može zauzeti polje ako na tom polju već nije figura iste boje. Ako figura stane na polje na kojem je protivnička figura, kaže se da je ta figura zarobljena ili pojedena, te ju se uklanja iz igre.

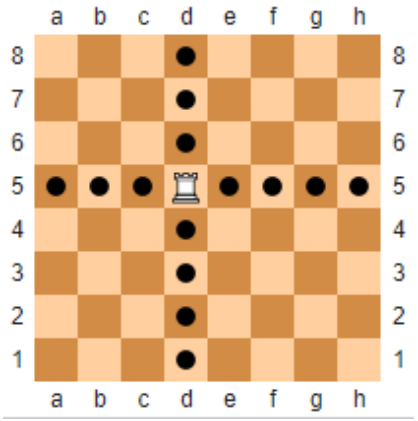
Pješak se može kretati jedno polje prema naprijed, a ako se prvi put pomiče moguće ga je pomaknuti i dva mjesta prema naprijed pod uvjetom da su oba mjesta prazna. Pješak može uzeti protivničku figuru ako mu se ona nalazi na dijagonali ispred njega (Slika 2.2.).



Slika 3.2. Mogući potezi pješaka

Uz pješaka se vežu i dva posebna poteza: *en passant* i promocija. Promocija pješaka se događa kada dospije do posljednjeg reda i tada ga se može zamijeniti s bilo kojom jačom figurom osim kralja. *En passant* je potez u kojem pješak može pojesti protivničkog susjednog pješaka nakon što potonji napravi dvostruki skok, ako ga je mogao pojesti da se pomaknuo za jedno polje. Iako najbrojnija i najslabija figura u šahu, pješak je ključna figura u početnoj fazi partije te konačni rezultat uvelike ovisi o njihovoj adekvatnoj organizaciji.

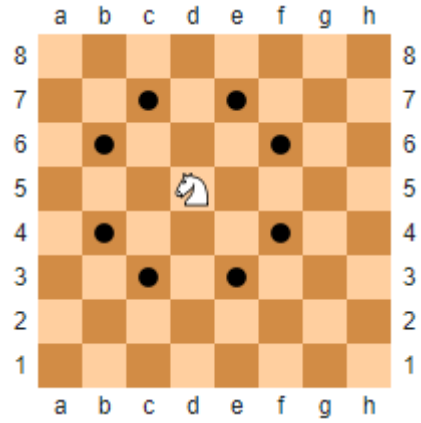
Top ili kula je figura koja se može kretati samo po redu i stupcu na kojem se nalazi te ne može preskakati figure (Slika 2.3.). Što je usamljeniji u svom retku ili stupcu to top ima veći utjecaj na partiju. Top sudjeluje i u posebnom potezu s kraljem koji se naziva rokada (rošada).



Slika 3.3. Mogući potezi topa

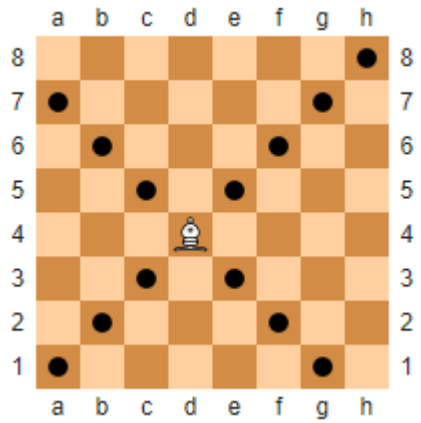
Skakač (konj) je figura koja se kreće u obliku slova L, odnosno dva polja u jednom smjeru i jedno polje u smjeru okomitom na smjer u kojem je išao prva dva polja (Slika 2.4.) te stoga može imati maksimalno osam mogućih poteza i uz pješaka je jedina figura koja se može pomicati u prvom

potezu. Skakač je jedinstvena figura po tome što može preskakati druge figure, a zbog svojih karakteristika najutjecajniji je blizu sredine ploče.



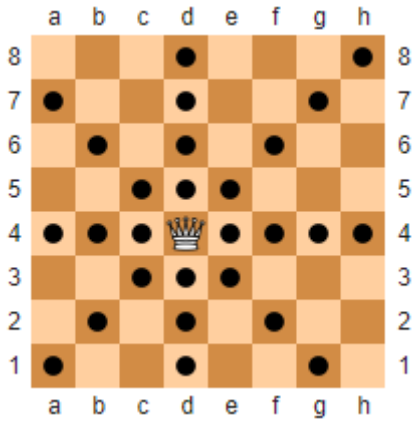
Slika 3.4. Mogući potezi skakača

Lovac se kreće ukoso, a kao i ostale figure osim skakača, ne može preskakati druge figure (Slika 2.5.). Svaki lovac ograničen je na kretanje samo po poljima iste boje. Iz tog razloga su osjetno korisniji kada se koriste u paru. Najbolja pozicija za lovca je u blizini dijagonale gdje može upravljati velikim prostorom, ali da je dovoljno udaljen od protivničkih figura [5].



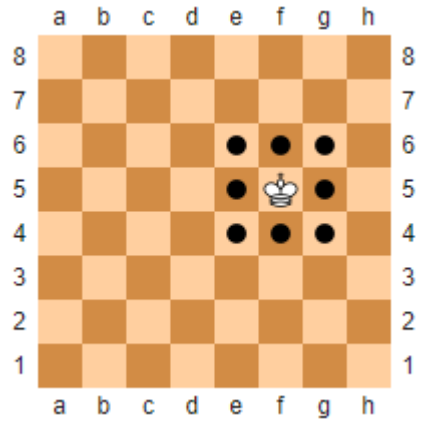
Slika 3.5. Mogući potezi lovca

Kraljica ili dama najmoćnija je figura u šahu. Može se kretati ukoso te po retku i stupcu na kojem se nalazi (Slika 2.6.). Budući da je najjača napadačka figura istovremeno je i najvrjednija. Stoga ju je dobro držati na poziciji s koje ima velik utjecaj na ploči, ali i na kojoj ju ne ugrožavaju protivničke figure. Kod promocije pješak se gotovo isključivo zamjenjuje kraljicom.



Slika 3.6. Mogući potezi kraljice

Kralj se može kretati u svim smjerovima za jedno mjesto (Slika 2.7.). Ako se kralj nalazi u šah poziciji, u sljedećem potezu njegov igrač mora se obraniti ili pomicanjem kralja ili zarobljavanjem figure u napadu ili blokadom figure u napadu. Ako takav potez nije moguć, onda se radi o „šah matu“, tj. igra je izgubljena. Poseban potez zvan rokada se izvodi ako između kralja i topa nema nijedne figure, a oni još nisu pomaknuti, a nužno je i da nijedno od polja kroz koja bi kralj prošao nije pod šahom. Tada se kralja pomiče dva polja prema topu, a topa se premješta iza kralja prema sredini. Rokada se smatra izrazito korisnim potezom u razvijanju figura koji istovremeno sklanja kralja na sigurno i omogućava topu dolazak na optimalnu poziciju [6].



Slika 3.7. Mogući potezi kralja

4. KORIŠTENE TEHNOLOGIJE

4.1. Visual Studio

Microsoft Visual Studio integrirano je programsko okruženje (eng. *IDE*) u vlasništvu tvrtke Microsoft i koristi se za razvoj računalnih programa, web usluga i mobilnih aplikacija. Okruženje podržava brojne programske jezike uključujući C, C++, C#, Python, JavaScript, itd. kao i alate za upravljanje izvornim kodom (Git).

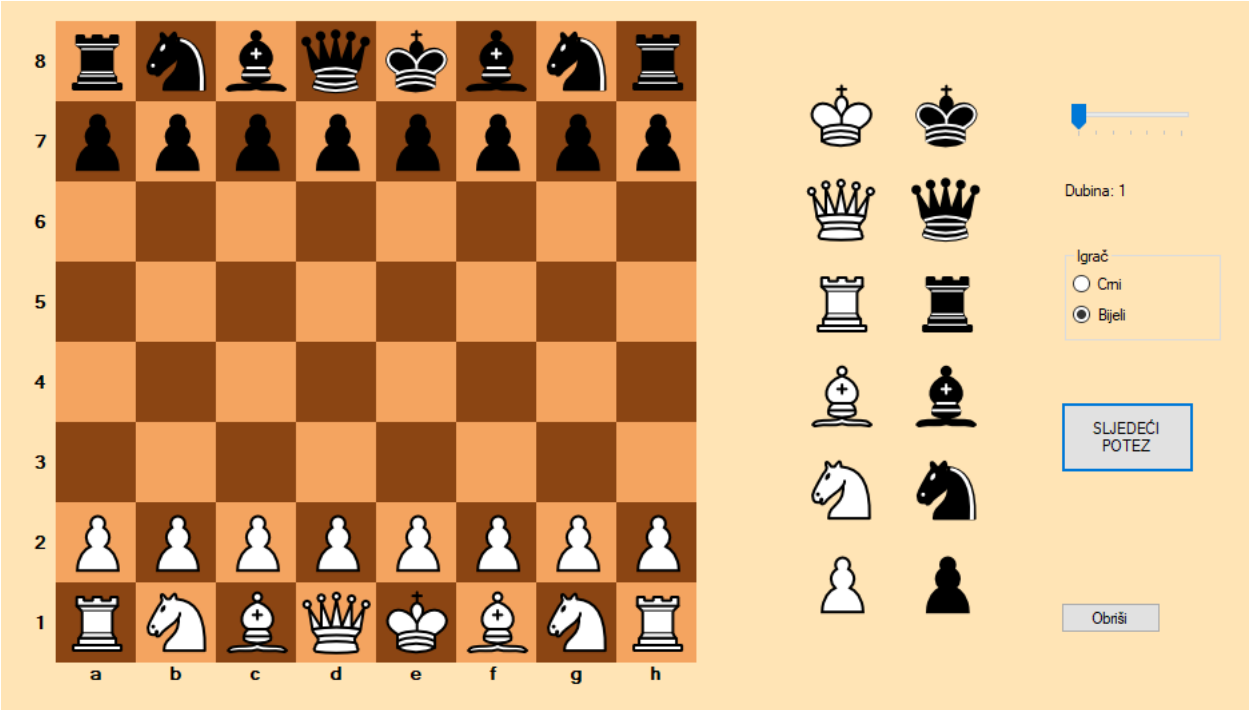
4.2. Programski jezik C#

C# (*C sharp*) moderni je programski jezik visoke razine kojeg je 2000. godine razvio Microsoft kao dio .NET platforme. C# je objektno orijentirani jezik što omogućava razvoj programskih rješenja temeljenih na objektima sa svojstvima i metodama. Podržan je jednostavan rad s iznimkama, rad s kolekcijama, a upravljanje memorije je automatsko. Sustav za prikupljanje smeća (eng. *Garbage collector*) upravlja korištenom memorijom u programima što znači da programer ne mora eksplicitno brinuti o oslobađanju memorije, nego se za to brine *garbage collector* koji prati koji se objekti trenutno koriste. Automatsko upravljanje memorijom sprječava brojne moguće probleme kao što su curenje memorije ili pristupanje zauzetoj memoriji [7].

4.3. Windows Forms

Windows Forms besplatna je biblioteka za grafičko korisničko sučelje uključena u .NET programsko okruženje. Korištena je za stvaranje efikasnih *desktop* aplikacija za operacijski sustav Windows.

Za potrebe programskog rješenja implementirana je Windows Forms aplikacija radi jednostavnog korištenja algoritma za generiranje poteza u šahu.



Slika 4.1. Grafičko korisničko sučelje

Korisnik može na ploči namjestiti situaciju po volji. Klikom miša na figure pored ploče korisnik ih može duplicirati na ploči. Figure koje se već nalaze na ploči ne dupliciraju se nego se pomiču na odabranu lokaciju. Kada je određena figura odabrana, pozadinska boja tog polja označena je sivom bojom radi lakšeg snalaženja. Moguće je namjestiti i dubinu pretrage stabla na kliznoj traci te odabrati koji je igrač na redu za potez. Klikom na gumb „SLJEDEĆI POTEZ“ ispisuje se potez koji je računalo odabralo kao najbolji. Ako se ne može odigrati nijedan potez ispisuje se „NEMA POTEZA“. Klikom na gumb „Obriši“ brišu se sve figure na šahovskoj ploči (Slika 3.1.).

5. IMPLEMENTACIJA ŠAHOVSKOG ALGORITMA

Šah je igra potpune informacije, odnosno među igračima nema nasumičnih niti skrivenih elemenata, a oba igrača počinju s jednakim uvjetima s izuzetkom da bijeli uvijek kreće prvi. Šah je i igra nulte sume što u teoriji znači da vrijedi da je gubitak jedne strane jednak dobitku druge strane i obrnuto. Matematički se može zapisati kao

$$u_1(\omega) + u_2(\omega) = 0$$

$$u_1(\omega) = -u_2(\omega),$$

za svaki potez odnosno ishod ω iz skupa Ω, gdje su $u_1 : \Omega \to \mathbb{R}$ i $u_2 : \Omega \to \mathbb{R}$ dobitak prvog odnosno drugog igrača [8].

5.1. Reprezentacija ploče

Ploča je izražena kao 8×8 matrica *char* tipa podatka. Velikim slovima označene su bijele figure a malima crne figure. Sukladno šahovskoj notaciji slovom K označen je kralj, Q je dama, R top, B lovac, N skakač te P pješak (engl. *K - king, Q - queen, R - rook, B - bishop, N - knight, P - pawn*). Izvornim kodom 4.1. prikazano je početno stanje na ploči:

```
private static char[,] board = new char[8, 8]
        {
            {'r','n','b','q','k','b','n','r'},
            {'p','p','p','p','p','p','p','p'},
            {' ',' ',' ',' ',' ',' ',' ',' '},
            {' ',' ',' ',' ',' ',' ',' ',' '},
            {' ',' ',' ',' ',' ',' ',' ',' '},
            {' ',' ',' ',' ',' ',' ',' ',' '},
            {'P','P','P','P','P','P','P','P'},
            {'R','N','B','Q','K','B','N','R'}};
```

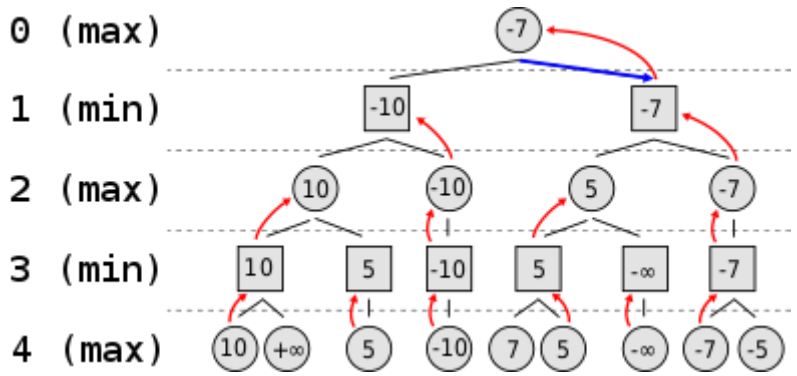
Izvorni kod 4.1. Reprezentacija ploče

5.2. Funkcija pretrage

Funkcija pretrage predstavlja algoritam koji će s obzirom na stanje na šahovskoj ploči pretražiti stablo igre, odnosno sve poteze koji se mogu odigrati u budućnosti do određene dubine. Cilj funkcije pretrage je pronaći optimalan potez uzevši u obzir situaciju na ploči.

5.2.1. Minimax

Najpoznatiji i najčešće korišteni algoritam za određivanje najboljeg poteza je *minimax* algoritam. Osnovni princip *minimax* algoritma je minimizacija maksimalnog gubitka za igrača koji je na potezu, odnosno određivanje poteza kojim će se igrača dovesti do minimalnog gubitka čak i kad bi protivnik igrao savršeno. U teoriji bi *minimax* mogao pretražiti sve moguće poteze i vratiti vrijednost između pobjede, neriješenog rezultata i poraza za tog igrača, no iz praktičnih razloga takav program nije moguć zbog eksponencijalne vremenske složenosti algoritma ($O(a^n)$ gdje je a broj izvedivih poteza, odnosno faktor grananja, a n dubina pretrage stabla). Stoga je osnovni cilj što bolje optimizirati, tj. skratiti vrijeme potrebno za izvršavanje algoritma jer se jedino time može postići veća dubina pretrage u razumnom vremenu.



Slika 5.1. Minimax

Na slici 4.1. ilustriran je primjer funkcioniranja *minimax* algoritma gdje čvorovi predstavljaju vrijednost statičke evaluacije odnosno stanje na ploči dok bi bridovi predstavljali poteze koji se mogu odigrati. Ono što je vidljivo je da algoritam kreće računati vrijednosti od listova, odnosno od najveće dubine do koje dolazi te naizmjenično prosljeđuje najmanje odnosno najveće vrijednosti djece prema roditeljskom čvoru te ponavlja isti postupak sve do korijena stabla. Može se primijetiti da dobivena vrijednost nije najbolja moguća od svih poteza, ali predstavlja najbolju poziciju do koje se sigurno može doći čak i kada bi protivnik uvijek odigrao svoj najbolji potez. Iako na slici nije prikazano, lako se može uočiti koliko brzo dolazi do izrazito velikog broja izvedivih poteza budući da je faktor grananja stabla uglavnom veći od 20, a uobičajena vrijednost mu je između 30 i 40, tj. svaki čvor u prosjeku ima toliki broj djece. Tom brzinom algoritam će već pri pretraživanju do dubine od četiri poteza morati pretražiti gotovo milijun mogućih situacija na šahovskoj ploči te je za veće dubine nepraktičan što pokazuje nužnost pravilne optimizacije

algoritma. Vidljivo je da algoritam prati tzv. „*depth-first search*“ uzorak, odnosno prioritizira obilazak stabla po dubini dok ne dođe do terminalnog čvora koji ovisi o dubini pretrage.

```
public static string Minimax(int depth, string move, int alpha, int beta,
int maximizingPlayer)
        {
             List<string> availableMoves = GetPossibleMoves();
            if (depth == 0 || availableMoves.Count.Equals(0))
            {
                return move + (Evaluator.Evaluate(availableMoves.Count, depth) *
maximizingPlayer).ToString();
            }
            if (depth > 1)
                OrderMoves(availableMoves, depth);
            maximizingPlayer*=-1;
            if (maximizingPlayer == -1)
            {
                for (int i = 0; i < availableMoves.Count; i++)
                {
                    Move(availableMoves[i]);
                    SwitchPlayer();
                    string oppositeMove = Minimax(depth - 1, availableMoves[i], alpha,
beta, maximizingPlayer);
                    int moveValue = (!oppositeMove.Contains('Q')) ?
int.Parse(oppositeMove.Substring(5)) : int.Parse(oppositeMove.Substring(6));
                    SwitchPlayer();
                    Undo(availableMoves[i]);
                    if (moveValue <= beta)
                    {
                        beta = moveValue;
                        if (depth == DEPTH)
                        {
                            move = (!oppositeMove.Contains('Q')) ?
oppositeMove.Substring(0, 5) : oppositeMove.Substring(0, 6);
                        }
                    }
                    if (alpha >= beta)
                    {
                        return move + beta;
                    }
                }
            }
            else if (maximizingPlayer == 1)
            {
                for (int i = 0; i < availableMoves.Count; i++)
                {
                    Move(availableMoves[i]);
                    SwitchPlayer();
                    string oppositeMove = Minimax(depth - 1, availableMoves[i], alpha,
beta, maximizingPlayer);
                    int moveValue = (!oppositeMove.Contains('Q')) ?
int.Parse(oppositeMove.Substring(5)) : int.Parse(oppositeMove.Substring(6));
                    SwitchPlayer();
                    Undo(availableMoves[i]);
                    if (moveValue > alpha)
                    {
                        alpha = moveValue;
                        if (depth == DEPTH)
{
                            move = (!oppositeMove.Contains('Q')) ?
oppositeMove.Substring(0, 5) : oppositeMove.Substring(0, 6);
```

```
                        }
                    }
if (alpha >= beta)
                    {
                        return move + alpha;
                    }
                }
            }
            return (maximizingPlayer == -1) ? move + beta : move + alpha;
}
```

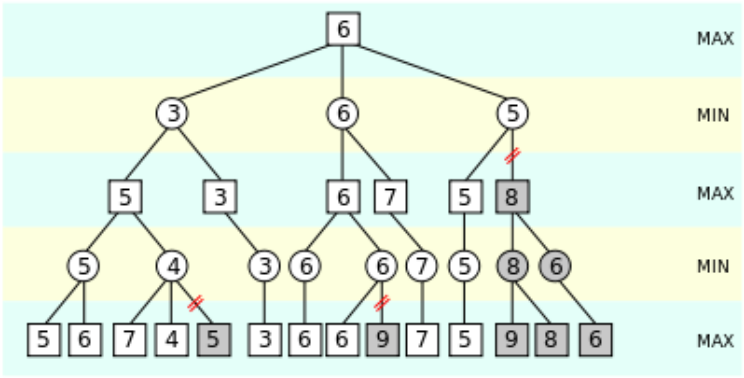
Izvorni kod 4.2. Minimax algoritam

Minimax algoritam prvo izračunava sve izvedive poteze i provjerava je li dubina pretrage 0, odnosno je li stablo doseglo do lista, ako jest, funkcija vraća potez i evaluaciju ploče u *string* obliku. Ako dubina nije 0 funkcija se izvršava dalje. Izvedivi potezi optimiziraju se sortiranjem ako je stablo barem na minimalnoj dubini na kojoj je ta provedba vremenski isplativa. Zatim se nakon provjere igrača provjerava svaki potez tako da ga se odigrava te zatim rekurzivno predaje protivniku dalje na igranje nakon kojeg se potez poništava. Ovaj dio ključan je u kreiranju stabla poteza ranije prikazanog. Zadnji je dio α – β rezanje, optimizacijska tehnika opisana u nastavku.

5.3. Optimizacijske tehnike

5.3.1. α – β rezanje

α – β rezanje (eng. *α – β pruning*) optimizacijska je tehnika kojom se reže velik dio *minimax* stabla, odnosno značajno skraćuje vrijeme izvođenja algoritma, ali nema nikakvog utjecaja na konačnu vrijednost koju će *minimax* vratiti. α – β rezanje sadrži dvije vrijednosti α i β koje predstavljaju minimalni gubitak za igrača na potezu odnosno maksimalni dobitak za protivnika, a njihove vrijednosti ovise o tome na kojoj dubini stabla se trenutno pretražuje [9].



Slika 5.2. *Alpha-Beta* rezanje

Na slici 4.2. prikazan je primjer funkcioniranja α – β rezanja gdje se prekida pretraga onda kada je sigurno da se ne može dobiti bolja vrijednost (u slučaju kada traži minimalnu vrijednost između 7, 4 i 5, te kada minimalna vrijednost postaje 4, jasno je da taj čvor više neće biti veći od 4, a budući da se na razini niže traži maksimalna vrijednost, nemoguće je nadmašiti broj 5 iz susjednog čvora te se pretraživanje prekida). Na ilustraciji ipak optimizacija ne dolazi do izražaja u toj mjeri kao u šahu zbog relativno malog omjera između faktora grananja i dubine stabla, dok je u šahu taj omjer puno veći zbog velikog broja mogućih ishoda te je u tom slučaju ova optimizacijska metoda puno isplativija. Isplativost α – β rezanja ovisi i o rasporedu ishoda u stablu, odnosno koliko će brzo naići na najbolji potez na određenoj razini i zato se u kombinaciji često koristi i optimizacija sortiranjem poteza koja dodatno povećava korist α – β rezanja. U najboljem slučaju ovakva optimizacija može smanjiti vremensku složenost obilaska stabla:

$$O(a^n) \rightarrow O(a^{n/2})$$

što znači da će za približno isto vrijeme gotovo udvostručiti dubinu pretrage stabla izvedivih poteza [4].

Tablica 4.1. Usporedba broja pretraženih poteza

Dubina n	a^n	$2a^{n/2}-1$
0	1	1
1	40	40
2	1,600	79
3	64,000	1,639
4	2,560,000	3,199
5	102,400,000	65,569
6	4,096,000,000	127,999
7	163,840,000,000	2,623,999

Na tablici 4.1. [10] prikazan je očekivani broj pretraživanja bez i sa α – β rezanjem pri vrijednosti a=40 te je vidljiv utjecaj na produbljivanje stabla pretrage. Za navedeni slučaj vrijedi maksimalna efikasnost optimizacije α – β rezanjem.

5.3.2. Transpozicijska tablica

Korištenjem *minimax* algoritma pretražuju se sve moguće situacije na šahovskoj ploči do određene dubine što je izrazito iscrpan i spor proces. Tijekom tog pretraživanja vrlo je vjerojatno da će program više puta naići na jednaku situaciju na ploči koju je ranije već sreo, ali drugačijim redoslijedom odigranih poteza. Ta pojava naziva se transpozicija. Budući da računalo nije u stanju samostalno odrediti je li određena situacija već riješena, koriste se transpozicijske tablice (*hash* tablice) [11]. Cilj je svaki poredak na šahovskoj ploči prikazati jedinstvenom brojčanom vrijednosti koja se pridodaje tablici. U slučaju ponovne pojave određene situacije, program će primijetiti da u tablici već postoji ta vrijednost te će ta situacija biti zabilježena. Tako se skraćuje broj pregledanih situacija te time ubrzava algoritam. Ipak, zbog izrazito velikog broja pregledanih poteza moguća je pojava kolizija, odnosno situacije u kojoj različiti rasporedi figura daju jednaku brojčanu vrijednost te se tada program može ponašati nepredvidivo te donijeti krivu odluku. U pravilu, pri većim dubinama pretrage, vremenska skraćenost koja se postiže, nadmašuje važnost mogućeg nepotrebnog rezanja stabla poteza te se iz tog razloga tablica i koristi.

Pri pokretanju programa inicijalizira se jedinstvena 8x8 matrica popunjena nasumičnim cijelim brojevima, gdje svaki broj predstavlja jedno polje. Metoda *CheckHashValue* poziva se na svakoj situaciji koja se obrađuje. Iterira se kroz cijelu ploču te se bilježi gdje se koja figura nalazi te se ovisno o figuri vrijednost nasumične matrice množi s odgovarajućim prostim brojem i pribraja ukupnoj vrijednosti. Na kraju izvođenja provjerava se postoji li ta vrijednost u *hash* tablici. Ako ne postoji, tada se broj dodaje tablici i funkcija vraća istinitu vrijednost, a u slučaju podudaranja funkcija vraća neistinitu vrijednost. Metoda se poziva unutar metode za sortiranje poteza *OrderMoves* koja je objašnjena u nastavku.

```
public static bool CheckHashValue()
        {
            int hashValue = 0;
            for (int i = 0; i < 8; i++)
            {
                for (int j = 0; j < 8; j++)
                {
                    switch (Engine.board[i, j])
                    {
                        case 'P':
                            hashValue += 1 * randomTable[i, j];

                            break;
                        case 'N':
                            hashValue += 2 * randomTable[i, j];

                            break;
                        case 'B':
                            hashValue += 3 * randomTable[i, j];

                            break;
                        case 'R':
                            hashValue += 5 * randomTable[i, j];

                            break;
                        case 'Q':
                            hashValue += 7 * randomTable[i, j];

                            break;
                        case 'K':
                            hashValue += 11 * randomTable[i, j];

                            break;
                        case 'p':
                            hashValue += 13 * randomTable[i, j];

                            break;
                        case 'n':
                            hashValue += 17 * randomTable[i, j];

                            break;
                        case 'b':
                            hashValue += 19 * randomTable[i, j];

                            break;
                        case 'r':
                            hashValue += 23 * randomTable[i, j];

                            break;
                        case 'q':
                            hashValue += 29 * randomTable[i, j];
                            break;
                        case 'k':
                            hashValue += 31 * randomTable[i, j];
                            break;
                    }
                }
            }
```

```
            if (hashSet.Contains(hashValue))
                return true;
            hashSet.Add(hashValue);
            return false;
        }

            hashSet = new HashSet<int>();
            randomGenerator = new Random();
            randomTable = new int[8, 8];
            for (int i = 0; i < 8; i++)
            {
                for (int j = 0; j < 8; j++)
                {
                    randomTable[i, j] =
randomGenerator.Next(int.MinValue/1000, int.MaxValue/1000);
                }
            }
```

Izvorni kod 4.3. CheckHashValue metoda

5.3.3. Sortiranje poteza

Vrijeme izvođenja algoritma s α – β rezanjem znatno ovisi o tome kako su svi potezi raspoređeni u stablu pretrage te se stoga ne može za svaku situaciju predvidjeti koliko će se dugo algoritam izvršavati. Optimizacija koja je korisna u skraćivanju vremena izvođenja pretrage je sortiranje poteza. Time se pri svakom pozivanju *minimax* pretrage na početku potezi određuju tako da se najbolji potezi provjeravaju prvi. To dovodi do značajnog smanjenja obrađenih poteza, odnosno povećanja odrezanih poteza. Iako i sam proces sortiranja koristi nezanemariv dio vremena, u većini slučajeva isplativ je zbog izrazito velikog broja poteza koji se ovim postupkom odbacuju.

```
private static void OrderMoves(List<string> moves, int depth)
        {
            List<int> values = new List<int>();
            List<Move> sortedMoves = new List<Move>();
            for (int i = 0; i<moves.Count; i++)
            {
                Move(moves[i].Substring(0, 5));
                if (depth > 2)
                {
                    if (Evaluator.CheckHashValue())
                    {
                        Undo(moves[i]);
                        continue;
                    }
                }
                values.Add(Evaluator.Evaluate(-1, 0));
                sortedMoves.Add(new Move(moves[i], values[values.Count-
1]));
                Undo(moves[i].Substring(0, 5));
            }
            for (int i = 0; i< sortedMoves.Count; i++)
            {
                for (int j = 0; j < sortedMoves.Count - 1; j++)
                {
                    if(sortedMoves[j].GetValue() >
                        sortedMoves[j + 1].GetValue())
                    {
                        Move temp = sortedMoves[j + 1];
                        sortedMoves[j + 1] = sortedMoves[j];
                        sortedMoves[j] = temp;
                    }
                }
            }
            moves.Clear();
            for(int i = 0; i<sortedMoves.Count; i++)
            {
                moves.Add(sortedMoves[i].GetMove());
            }
        }
```

Izvorni kod 4.4. Sortiranje poteza

5.4. Funkcija evaluacije

Funkcija evaluacije glavni je dio šahovskog algoritma uz funkciju pretrage i zadužen je za određivanje najboljeg izvedivog poteza. Njena zadaća je analizirati stanje na šahovskoj ploči, odnosno broj figura na ploči kao i njihovu poziciju, utjecaj na druge figure i mnoge druge atribute, ovisno o tipu i složenosti funkcije. Slabost evaluacijske funkcije je nedostatak intuitivnosti i strateškog razmišljanja koje imaju ljudi. Računalo nije sposobno na temelju jednog pogleda na ploču zaključiti na što se treba fokusirati, koje parametre zanemariti, a na koje se usredotočiti, ali čak i naizgled trivijalna evaluacijska funkcija u kombinaciji s dubokom funkcijom pretrage može

proizvesti zadovoljavajuće rezultate zbog evidentne prednosti u brzini obrade informacija u odnosu na ljudske igrače.

```
public static int Evaluate(int count, int depth)
        {
            int evaluationValue = EvaluatePieces(count);
            Engine.SwitchPlayer();
            evaluationValue -= EvaluatePieces(count);
            Engine.SwitchPlayer();
            if (count == 0)
              if (!Engine.IsKingSafe())
                  evaluationValue -= 20000 * depth;
           return evaluationValue * -1;
        }
```

Izvorni kod 4.5. Funkcija evaluacije

Kako bi se pravilno evaluiralo stanje na ploči potrebno je izračunati situaciju iz perspektive jednog igrača te tu vrijednost oduzeti od vrijednosti evaluacije protivničkog igrača. Time se traži potez koji će stvoriti najveću prednost u odnosu na protivnika. Provjerava se je li broj mogućih poteza jednak nuli u slučaju moguće šah mat ili pat situacije (neriješen ishod). U tom slučaju oduzima se velika vrijednost jer je to najgori ishod za igrača. Ploča se analizira na temelju tri parametra: materijalna i pozicijska evaluacija te evaluacija mobilnosti figura.

5.4.1. Materijalna evaluacija

Osnovna mjera situacije na šahovskoj ploči je broj pojedinih figura koje se nalaze na njoj. Materijalna evaluacija zbraja sve figure na ploči pri čemu su za jednog igrača vrijednosti pozitivne, a za drugog negativne.

Tablica 4.2. Materijalne vrijednosti figura

Pješak	100
Skakač	300
Lovac	300
Top	500
Kraljica	900
Kralj	9000

Vrijednosti figura kao i ostalih čimbenika izražene su u *centipawn (cp)* mjernim jedinicama, gdje 1 *centipawn* iznosi stoti dio vrijednosti pješaka, a koristan je jer omogućava izbjegavanje razlomaka u evaluaciji. Omjer figura 1/3/3/5/9 prvi je uveo Claude E. Shannon u svojim radovima [2], a danas je generalno prihvaćen iako te vrijednosti nisu nužno najbolje u svakoj situaciji. Kraljica je po tome vrjednija od topa i lovca ili topa i skakača, ali ne od dva topa, lovac i skakač vrijede kao tri pješaka, a top kao lovac ili skakač s dodatna dva pješaka. Najveći predmet rasprave su lovac i skakač. Prednosti skakača su njegovo preskakanje figura i jačina u ranoj fazi partije dok lovac, kada je pravilno razvijen, može pokriti veće područje od skakača. Nedostatak lovca je što nikada ne može promijeniti boju polja na kojemu se nalazi, ali se zato dva lovca u paru smatraju izrazito korisnim te im se u evaluaciji daje mala prednost u odnosu na skakače, tj. vrijednosti između 300 i 350 po lovcu. Vrijednost kralja može biti proizvoljna vrijednost, ali je bitno da se ta vrijednost ne može usporediti s drugim figurama po važnosti, odnosno mora biti veća od zbroja svih ostalih što dovodi do toga da će evaluacijska funkcija izbjegavati poteze koji bi mogli dovesti do nepovoljne situacije u kojoj bi se kralj mogao naći u šahu. Vrijednosti figura prikazane su u tablici 4.2.

5.4.2. Pozicijska evaluacija

Iako je prisutnost figura i njihovo čuvanje najočiglednije mjerilo kvalitete situacije na šahovskoj ploči, drugi bitan faktor je raspored tih figura. Tehnika korištena za pozicijsku evaluaciju su *piece-square* tablice. Izražene su kao 8×8 matrice s cjelobrojnim vrijednostima i predstavljaju kvalitetu položaja za svaku figuru, tj. što je vrijednost polja za određenu figuru veća, to je ta pozicija kvalitetnija, odnosno omogućava bolju razvijenost i iskoristivost figure na šahovskoj ploči. Postoji mnogo vrsta takvih matrica, a njihove vrijednosti empirijski su dobivene iz do sada odigranih šahovskih partija vrhunskih igrača. U ovom radu korištene su tablice napisane u izvornom kodu 4.6. [12].

```
private static readonly int[,] pawnPosition = new int[8, 8]
        {
            { 0,  0,  0,  0,  0,  0,  0,  0},
            {50, 50, 50, 50, 50, 50, 50, 50},
            {10, 10, 20, 30, 30, 20, 10, 10},
            { 5,  5, 10, 25, 25, 10,  5,  5},
            { 0,  0,  0, 20, 20,  0,  0,  0},
            { 5, -5,-10,  0,  0,-10, -5,  5},
            { 5, 10, 10,-20,-20, 10, 10,  5},
            { 0,  0,  0,  0,  0,  0,  0,  0}};
private static readonly int[,] knightPosition = new int[8, 8]
        {
            {-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}};
private static readonly int[,] bishopPosition = new int[8, 8]
        {
            {-20,-10,-10,-10,-10,-10,-10,-20},
            {-10,  0,  0,  0,  0,  0,  0,-10},
            {-10,  0,  5, 10, 10,  5,  0,-10},
            {-10,  5,  5, 10, 10,  5,  5,-10},
            {-10,  0, 10, 10, 10, 10,  0,-10},
            {-10, 10, 10, 10, 10, 10, 10,-10},
            {-10,  5,  0,  0,  0,  0,  5,-10},
            {-20,-10,-10,-10,-10,-10,-10,-20}};
private static readonly int[,] rookPosition = new int[8, 8]
        {
            { 0,  0,  0,  0,  0,  0,  0,  0},
            {50, 50, 50, 50, 50, 50, 50, 50},
            {10, 10, 20, 30, 30, 20, 10, 10},
            { 5,  5, 10, 25, 25, 10,  5,  5},
            { 0,  0,  0, 20, 20,  0,  0,  0},
            { 5, -5,-10,  0,  0,-10, -5,  5},
            { 5, 10, 10,-20,-20, 10, 10,  5},
            { 0,  0,  0, 10, 10,  0,  0,  0}};
private static readonly int[,] queenPosition = new int[8, 8]
        {
            {-20,-10,-10, -5, -5,-10,-10,-20},
            {-10,  0,  0,  0,  0,  0,  0,-10},
            {-10,  0,  5,  5,  5,  5,  0,-10},
            { -5,  0,  5,  5,  5,  5,  0, -5},
            {  0,  0,  5,  5,  5,  5,  0, -5},
            {-10,  5,  5,  5,  5,  5,  0,-10},
            {-10,  0,  5,  0,  0,  0,  0,-10},
            {-20,-10,-10, -5, -5,-10,-10,-20}};
private static readonly int[,] kingPosition = new int[8, 8]
        {
            {-30,-40,-40,-50,-50,-40,-40,-30},
            {-30,-40,-40,-50,-50,-40,-40,-30},
            {-30,-40,-40,-50,-50,-40,-40,-30},
            {-30,-40,-40,-50,-50,-40,-40,-30},
            {-20,-30,-30,-40,-40,-30,-30,-20},
            {-10,-20,-20,-20,-20,-20,-20,-10},
            { 20, 20,  0,  0,  0,  0, 20, 20},
            { 20, 30, 10,  0,  0, 10, 30, 20}};
private static readonly int[,] kingEndPosition = new int[8, 8]
        {
            {-50,-40,-30,-20,-20,-30,-40,-50},
            {-30,-20,-10,  0,  0,-10,-20,-30},
            {-30,-10, 20, 30, 30, 20,-10,-30},
            {-30,-10, 30, 40, 40, 30,-10,-30},
            {-30,-10, 30, 40, 40, 30,-10,-30},
            {-30,-10, 20, 30, 30, 20,-10,-30},
            {-30,-30,  0,  0,  0,  0,-30,-30},
            {-50,-30,-30,-30,-30,-30,-30,-50}};
```

Izvorni kod 4.6. *piece-square* tablice

Vrijednost pozicijske evaluacije pribraja se vrijednosti materijalne evaluacije figura. Npr. pješak na polju (6, 4) (Indeksi počinju od 0. (6,4) predstavlja peti element u sedmom retku.) odnosno e2 po šahovskoj notaciji, vrijedi 100 + (-20), odnosno 80 *centipawna (cp)*, dok se njegovim pomakom na (4, 4), tj. e4 njegova vrijednost podiže na 100 + 20 odnosno 120 *cp*. Iz izvornog koda 4.6. vidljivo je da se ovim pristupom teži ranom uspostavljanju dominacije na središnjem dijelu šahovske ploče što je inače jedna od temeljnih strategija u šahu [13]. Nastoji se postići kontrola nad četiri središnja polja ranim razvojem pješaka te skakača i lovaca prema sredini, ranom rokadom dovesti kralja na sigurno i izbjeći prerano korištenje kraljice.

5.4.3. Evaluacija mobilnosti

Ako figure imaju više mogućnosti za pomicanje, igrač će imati više mogućnosti za odigravanje kvalitetnog poteza. Stoga se nagrađuje potez koji figuru dovodi u otvoreniju poziciju na ploči.

```
private static int EvaluatePieces(int count)
        {
            int piecesValue = 0;
            int positionValue = 0;
            int mobilityValue = count*5;
            bool bishopPair = false;
            int kingI = -1, kingJ = -1;
            int[] pawnsInLine = new int[8] { 0, 0, 0, 0, 0, 0, 0, 0 };
            for (int i = 0; i < 8; i++)
            {
                for (int j = 0; j < 8; j++)
                {
                    switch (Engine.board[i, j])
                    {
                        case 'P':
                            piecesValue += 100;
                            positionValue += pawnPosition[i, j];
                            pawnsInLine[j]++;
                            if(pawnsInLine[j] > 1)
                                positionValue -= 50;
                            if (Engine.IsInsideBounds(i - 1, j - 1))
                            {
                                if (Engine.board[i - 1, j - 1] == 'P')
                                    positionValue += 10;
                            }
                            if (Engine.IsInsideBounds(i - 1, j + 1))
                            {
                                if (Engine.board[i - 1, j + 1] == 'P')
                                    positionValue += 10;
                            }
                            break;
```

```
                    case 'N':
                        piecesValue += 300;
                        positionValue += knightPosition[i, j];
                        break;
                    case 'B':
                        positionValue += bishopPosition[i, j];
                        piecesValue += bishopPair ? 400 : 300;
                        bishopPair = !bishopPair;
                        break;
                    case 'R':
                        piecesValue += 500;
                        positionValue += rookPosition[i, j];
                        break;
                    case 'Q':
                        piecesValue += 900;
                        positionValue += queenPosition[i, j];
                        break;
                    case 'K':
                        piecesValue += 9000;
                        kingI = i;
                        kingJ = j;
                        break;
                }
            }
        }
        if (piecesValue >= 12000)
        {
            if(kingI == -1)
            {
                piecesValue -= 10000;
            }
            else
            {
                positionValue += kingPosition[kingI, kingJ];
            }
        }
        else
        {
            if (kingI == -1)
            {
                piecesValue -= 10000;
            }
            else
            {
                positionValue += kingEndPosition[kingI, kingJ];
            }
        }
        return piecesValue + positionValue + mobilityValue;
    }
```

Izvorni kod 4.7. Evaluacija materijala, pozicije i mobilnosti

Metoda *EvaluatePieces* vraća zbroj tri varijable koje predstavljaju tri ranije navedene evaluacije materijala, pozicije i mobilnosti. Parametar *count* predstavlja broj izvedivih poteza za igrača te o

njemu ovisi vrijednost evaluacije mobilnosti. Pozicijska evaluacija osim rasporeda figura na ploči prati i raspored pješaka preferirajući njihovo pravilno razvijanje koje je ključno za preuzimanje inicijative na ploči. Nagrađuje se postavljanje pješaka u dijagonalu gdje jedan drugog štite, a penalizira se njihovo dupliranje po stupcima. Prati se broj lovaca zbog njihove velike vrijednosti kada su korišteni u kombinaciji. Pozicioniranje kralja ovisi o stadiju igre. Što se igra više razvija kralja se potiče van rubnih područja radi smanjenja njegove ranjivosti.

6. REZULTATI

Testiranje programa izvodilo se na računalu s AMD Ryzen 5 3600 procesorom s radnim taktom od 3.59 GHz.

6.1. Vrijeme izvođenja

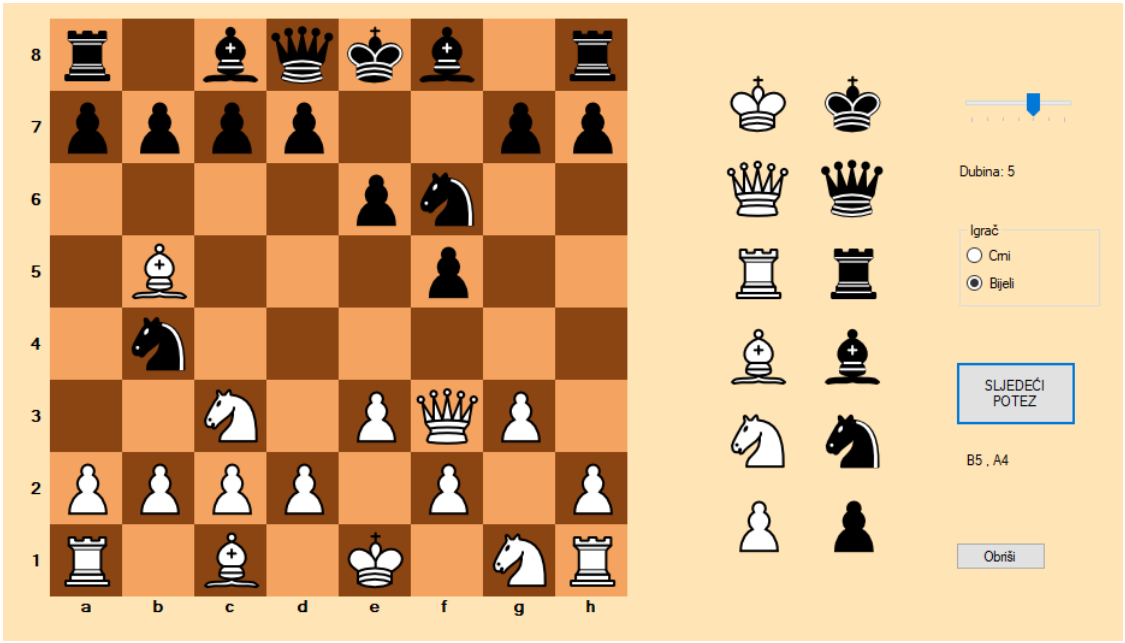
Vrijeme izvođenja testirano je na skupu od deset zagonetki sa stranice ChessManiac [14] s različitim dubinama pretrage. Uspoređeno je prosječno vrijeme izvođenja algoritma za slučajeve bez sortiranja poteza i sa sortiranjem poteza pri različitim dubinama pretrage (Tablica 5.1.).

Tablica 5.1.. Prosječno vrijeme izvođenja algoritma

Dubina pretrage	bez sortiranja [min:sek.tis]	sa sortiranjem [min:sek.tis]
3	00:00.042	00:00.032
4	00:00.259	00:00.115
5	00:03.329	00:01.047
6	00:22.743	00:04.072
7	04:32.325	00:28.471

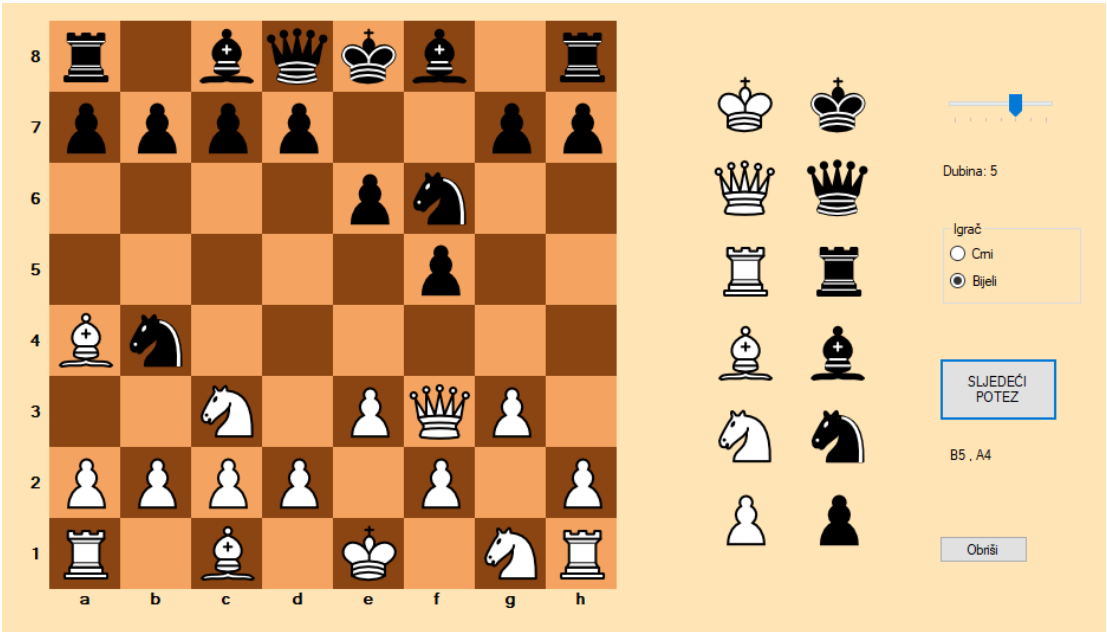
Iz navedenih rezultata vidljivo je značajno skraćenje vremena izvođenja algoritma kada se koristi sortiranje poteza. Tom metodom maksimalno se iskorištava potencijal $\alpha - \beta$ rezanja jer se najbolji potezi obrađuju prvi te se provjerava znatno manji broj situacija nego kod nasumičnog redoslijeda. Unutar metode za sortiranje poteza poziva se i metoda za provjeru transpozicijske tablice što dodatno ubrzava algoritam, dok se konačne vrijednosti koje vraća funkcija pretrage ne mijenjaju. Pretrage na dubini do pet poteza izvršavaju se praktički trenutačno, unutar dvije sekunde. Pretraga na dubini od šest poteza izvršava se uglavnom do deset sekundi, a na sedam poteza unutar minute. Vrijeme izvođenja prvenstveno ovisi o fazi partije. Algoritam je najbrži u fazi otvaranja te u fazi završnice kada je manji broj mogućih poteza dok je u pravilu sporiji u središnjoj fazi, kada je broj mogućih poteza najveći zbog velikog broja razvijenih figura.

6.2. Primjer rada

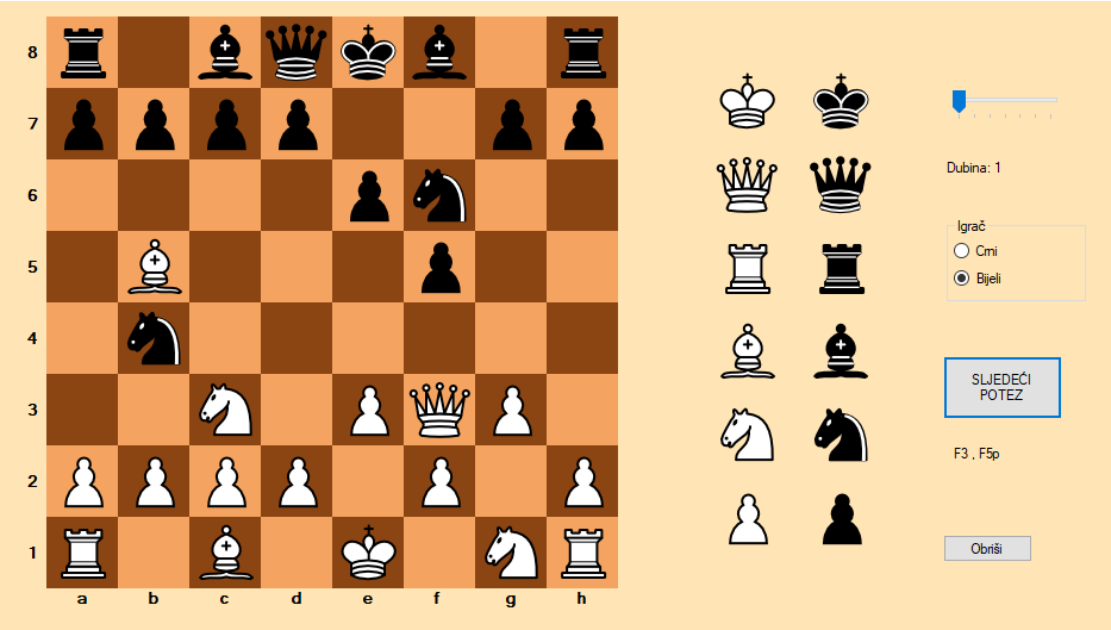


Slika 6.1. Određivanje poteza pri dubini 5

U prikazanoj situaciji (Slika 5.1.) bijeli igrač je u opasnosti da izgubi pješaka i topa u četiri poteza ako ne zaštiti polje c2 ili kralja na e1. Dubina pretrage je pet poteza. Kao rezultat, lovac se pomiče na polje a4 te blokira pokušaj protivnika (Slika 5.2.).

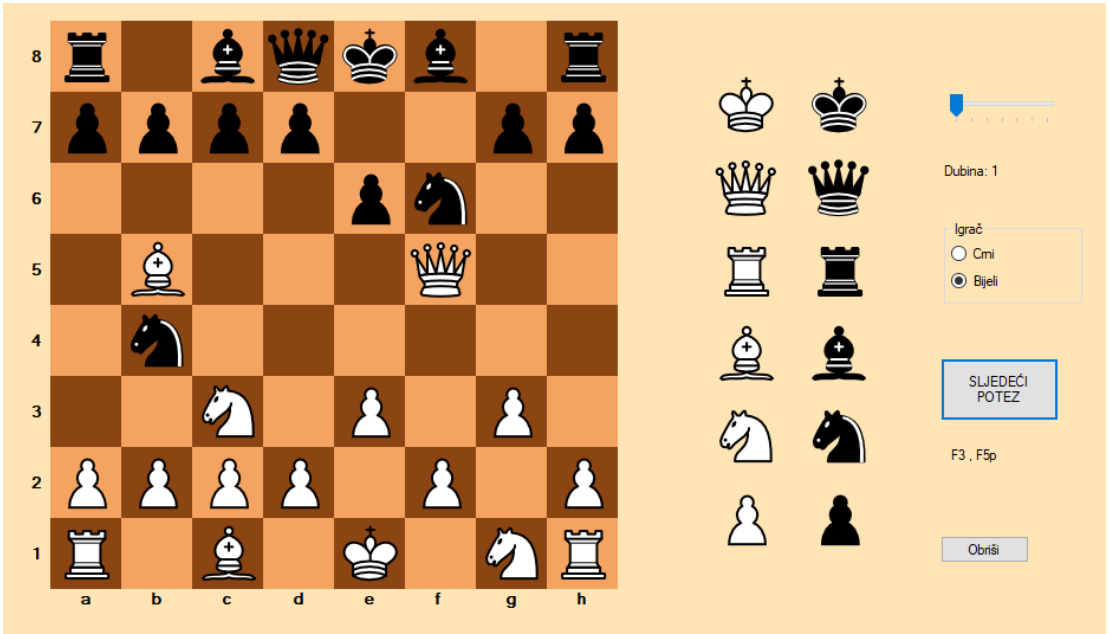


Slika 6.2. Odigran potez pri dubini 5



Slika 6.3. Određivanje poteza pri dubini 1

Na slici 5.3. prikazana je ista situacija, ali je dubina pretrage ovaj put postavljena na jedan. Rezultat toga je nelogično uzimanje pijuna kraljicom na f5 (Slika 5.4.).



Slika 6.4. Odigran potez pri dubini 1

Budući da u ovom slučaju računalo ne vidi situaciju nakon prvog poteza bijelog igrača, na temelju evaluacijske funkcije zaključuje da je to najbolji potez jer se uzima protivnički pješak koji vrijedi 100 *cp*. Ta situacija kada je opasna situacija izvan dosega algoritma naziva se efekt horizonta. Navedeni primjer dobar je pokazatelj važnosti dubine pretrage poteza, ali također pokazuje i neke slabosti ovog računalnog programa.

Iako je sposobno odraditi milijarde instrukcija u sekundi, računalo i dalje nema sposobnost strateškog razmišljanja. Implementirani program nema informacije o ranije odigranim potezima što može dovesti do pat situacije uzrokovane ponavljanjem istih poteza. To je posljedica statičke evaluacijske funkcije koja ne poznaje kontekst partije nego samo evaluira situaciju na ploči. Drugi problem koji je uzrokovan tim nedostatkom je prisutan pri mogućem uzimanju protivničkog kralja nakon određenog broja poteza. Tijekom odvijanja tih poteza algoritam nastavlja pretraživanje po istoj dubini te može odbiti završavanje partije jer mu je garantirana druga situacija s boljom evaluacijskom vrijednosti. Posljedica toga je da algoritam preferira protivničke figure ispred mogućeg završetka partije. Poboljšanja na tom području značajno bi unaprijedila igru u završnici. Još jedan problem koji je moguće riješiti je efekt horizonta. On je posljedica ograničenja u dubini pretrage stabla na fiksne vrijednosti, a pojavljuje se kada pretraga poteza dosegne maksimalnu dubinu. Tada se računalu čini da je potez isplativ, jer ne može vidjeti idući protivnikov potez koji može biti i uzimanje kraljice [15] kao što je prikazano na slici 5.2. Rješenje tog problema je *quiescence* pretraga u kojoj dubina stabla nije fiksna, nego se pretraga produžuje do veće dubine u situacijama koje su potencijalno opasne poput uzimanja protivničke figure.

Iako su prisutni nedostaci u implementaciji šahovskog algoritma, njegova prednost je brzina igranja, što je posebno izraženo u brzopoteznom (eng. *rapid chess*) i munjevitom (eng. *blitz chess*) šahu protiv ljudi koji nemaju sposobnost tako brze obrade informacija. Druga prednost je konzistencija algoritma jer je mala vjerojatnost da će učiniti kardinalnu grešku tijekom igre.

7. ZAKLJUČAK

Cilj rada bio je implementirati računalni program koji će na temelju zadane situacije na šahovskoj ploči generirati dobar potez u razumnom vremenu. Glavni problemi koji se pojavljuju pri rješavanju takvog problema su kako odrediti koja situacija je najpovoljnija te kako efikasno provjeriti sve moguće situacije. Rješenje problema su statička funkcija evaluacije koja obrađuje brojne parametre na ploči kako bi se dobila kvaliteta situacije te funkcija pretrage svih poteza temeljena na *minimax* algoritmu. Zbog eksponencijalne složenosti algoritam je optimiziran i može raditi do dubine od šest do sedam polupoteza unutar razumnog vremena. Program nema sposobnost strateškog razmišljanja niti poznaje kontekst izvan funkcije evaluacije što mu je slabost u odnosu na ljude. U radu su navedeni najbitniji nedostaci kao loša završnica i prisutan efekt horizonta te se ti problemi mogu riješiti budućim proširenjima. Unatoč nedostatcima program je znatno brži od ljudi i nije sklon pogreškama te je uz postojeću implementaciju dovoljno dobar da pruži dobar izazov amaterima.

LITERATURA

[1] J. McCarthy, Chess as the Drosophila of AI, Journal of the International Computer Chess Association no. 4, vol. 12, pp. 199-206 https://link.springer.com/chapter/10.1007/978-1-4613-9080-0_14

[2] C. E. Shannon, Programming a Computer for Playing Chess, Philosophical Magazine, Ser.7, Vol. 41, No. 314, pp. 256-275., ožujak, 1950.

[3] P. Bijl, A.P. Tiet, Exploring modern chess engine architectures, Victoria University, Melbourne, 2021.

[4] D. Kasak, Određivanje izvedivih poteza u šahu pomoću bitboard zapisa pozicije, Fakultet elektrotehnike, računarstva i informacijskih tehnologija Osijek, Osijek, 2017.

[5] Introduction to chess strategy, Chess Strategy Online, dostupno na: https://www.chessstrategyonline.com/content/tutorials/introduction-to-chess-strategy [kolovoz, 2021.]

[6] L. Watson, How to castle in chess: Our guide to mastering this special rule, Chessable, 2019., dostupno na: https://www.chessable.com/blog/how-to-castle-in-chess/ [lipanj, 2021.]

[7] Fundamentals of garbage collection, Microsoft Docs, studeni, 2019. dostupno na: https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals [srpanj, 2021.]

[8] Ken Binmore, Playing for Real A Text on Game Theory, Oxford University Press, Inc. New York, 2007.

[9] CS 161 Recitation Notes - Minimax with Alpha Beta Pruning, University of California, dostupno na: http://web.cs.ucla.edu/~rosen/161/notes/alphabeta.html [lipanj, 2021.]

[10] Alpha-Beta, Chess Programming Wiki, dostupno na: https://www.chessprogramming.org/Alpha-Beta [lipanj, 2021.]

[11] Transposition Table, Chess Programming Wiki, dostupno na: https://www.chessprogramming.org/Transposition_Table [srpanj, 2021.]

[12] Simplified Evaluation Function, Chess Programming Wiki, dostupno na: https://www.chessprogramming.org/Simplified_Evaluation_Function [lipanj, 2021.]

[13] I. Smirnov, Chess Opening Fundamentals By GM Igor Smirnov, Remote Chess Academy, prosinac, 2019.

[14] ChessManiac, dostupno na: http://www.chessmaniac.com/ELORating/ELO_Chess_Rating.shtml [kolovoz, 2021.]

[15] Quiescence Search, Chess Programming Wiki, dostupno na: https://www.chessprogramming.org/Quiescence_Search [kolovoz, 2021.]

SAŽETAK

U ovom radu opisani su teorijski principi šaha te je implementiran program koji na temelju aktualne situacije na šahovskoj ploči određuje najbolji potez. Heuristika programa temeljena je na pretpostavci da je svakoj situaciji na ploči moguće odrediti brojčanu vrijednost koja predstavlja kvalitetu poteza za igrača na redu. Tu vrijednost određuje statička funkcija evaluacije koja uzima u obzir broj, poziciju i mobilnost svih figura na ploči. Funkcija pretrage temeljena na *minimax* algoritmu stvara stablo svih okolnosti koje su moguće na ploči do određene dubine. Zbog eksponencijalne vremenske složenosti nužna je optimizacija pretrage. Objašnjene su implementacije $\alpha - \beta$ rezanja, sortiranja poteza i transpozicijske tablice koji značajno skraćuju vrijeme izvođenja programa. Testirane su performanse programa te su objašnjeni prisutni nedostaci u programu. Za jednostavnije korištenje programa implementirana je Windows Forms aplikacija s grafičkim korisničkim sučeljem.

Ključne riječi: algoritam, C#, funkcija evaluacije, minimax, šah

ABSTRACT

Chess Engine Algorithm in C#

This paper describes theoretical principles of chess and implements a program which generates the best move based on the current chess board situation. The heuristics of the program are based on the assumption that it is possible to determine a numerical value for each situation on the board, which represents the quality of a situation for the player on the move. That value is determined by the static evaluation function which takes into account the number, position and mobility of all the figures on the board. The search function based on the minimax algorithm creates a tree of all the circumstances that are possible on the board to a certain depth. Due to the exponential time complexity, search engine optimization is necessary. Implementations of $\alpha - \beta$ pruning, move ordering and transposition table which significantly reduce the program execution time are explained in the paper. The performance of the program is tested and its shortcomings are explained. For easier use of the program, a Windows Forms application with a graphical user interface has been implemented.

Key words: algorithm, chess, C#, evaluation function, minimax

ŽIVOTOPIS

David Kvesić rođen je 19. lipnja 1999. godine u Osijeku. 2014. godine završava osnovno obrazovanje u Osnovnoj školi Vladimira Becića. Nakon toga završava srednjoškolsko obrazovanje s odličnim uspjehom u I. Gimnaziji Osijek. Tijekom osnovnoškolskog i srednjoškolskog obrazovanja sudjelovao je na brojnim županijskim natjecanjima iz geografije, fizike, povijesti i njemačkog jezika. Višestruki je županijski prvak iz geografije. 2018. godine upisuje Preddiplomski sveučilišni studij računarstva na Fakultetu elektrotehnike, računarstva i informacijskih tehnologija Osijek. Radio je kao instruktor iz fizike u centru instrukcija Edukos. Na fakultetu je povremeno zaposlen kao demonstrator na brojnim kolegijima. Aktivno se služi engleskim jezikom.

_____

Potpis autora