Određivanje izvedivih poteza u šahu pomoću bitboard zapisa pozicije

Kasak, Denis

SVEUČILIŠTE JOSIPA JURJA STROSSMAYERA U OSIJEKU
ELEKTROTEHNIČKI FAKULTET

Sveučilišni diplomski studij

Određivanje izvedivih poteza u šahu pomoću bitboard zapisa pozicije

Diplomski rad

Denis Kasak

Osijek, 2017.

FERIT
FAKULTET ELEKTROTEHNIKE, RAČUNARSTVA I INFORMACIJSKIH TEHNOLOGIJA OSIJEK

Obrazac D1: Obrazac za imenovanje Povjerenstva za obranu diplomskog rada

Osijek, 21.09.2017.

Odboru za završne i diplomske ispite

Imenovanje Povjerenstva za obranu diplomskog rada

Ime i prezime studenta:	Denis Kasak
Studij, smjer:	Diplomski sveučilišni studij Računarstvo, smjer Procesno računarstvo
Mat. br. studenta, godina upisa:	D-609R, 13.10.2016.
OIB studenta:	84710985675
Mentor:	Prof.dr.sc. Željko Hocenski
Sumentor:	Doc.dr.sc. Ivan Aleksi
Sumentor iz tvrtke:	
Predsjednik Povjerenstva:	Doc.dr.sc. Ivan Aleksi
Član Povjerenstva:	Filip Sušac
Naslov diplomskog rada:	Određivanje izvedivih poteza u šahu pomoću bitboard zapisa pozicije
Znanstvena grana rada:	**Programsko inženjerstvo (zn. polje računarstvo)**
Zadatak diplomskog rada:	U ovom diplomskom radu potrebno je napisati program koji određuje skup izvedivih slijedećih poteza za određenu šahovsku poziciju. Potrebno ju je pozivati tijekom svakog polupoteza, odnosno kada odigra pojedini igrač. Potrebno je opisati bitboard i primijeniti ga uz korištenje bit-operacija.
Prijedlog ocjene pismenog dijela ispita (diplomskog 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: 3 bod/boda Razina samostalnosti: 3 razina
Datum prijedloga ocjene mentora:	21.09.2017.

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, 03.10.2017.

Ime i prezime studenta:	Denis Kasak
Studij:	Diplomski sveučilišni studij Računarstvo, smjer Procesno računarstvo
Mat. br. studenta, godina upisa:	D-609R, 13.10.2016.
Ephorus podudaranje [%]:	15

Ovom izjavom izjavljujem da je rad pod nazivom**:**
Određivanje izvedivih poteza u šahu pomoću bitboard zapisa pozicije

izrađen pod vodstvom mentora Prof.dr.sc. Željko Hocenski

i sumentora Doc.dr.sc. Ivan Aleksi

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

Problematika računala koja igraju šah već dugo zaokuplja računalne znanstvenike i inženjere zbog relativne jednostavnosti pravila igre, ali velike rezultantne kompleksnosti zbog kombinatorne eksplozije. Osnovni cilj računalnih šahovskih programa je, za danu poziciju, iz mogućih šahovskih poteza, odrediti dobar potez u razumnom vremenu. Kako bismo utvrdili koji potez je najbolji, partiju šaha možemo modelirati kao stablo čiji čvorovi predstavljaju pozicije (konfiguracije šahovske ploče), a bridovi poteze koji vode od jedne pozicije do druge. Krenuvši od dane pozicije, takvo stablo moguće je pretraživati kako bismo pronašli optimalan potez. Međutim, zbog ogromnog broja mogućih pozicija, egzaktno rješenje nije praktično izvedivo te je stoga polje implementacije šahovskih programa izuzetno bogato kompleksnim tehnikama i heuristikama koje omogućavaju dobre rezultate.

Kako bismo stablo igre uopće mogli pretražiti, potrebno je šahovsku ploču modelirati unutar računala, a zatim i za danu šahovsku poziciju generirati moguće poteze. Budući da je za praktičan šahovski program bitno da može odrediti dobar potez u razumnom vremenu, ove radnje moraju biti brzo izvedive. Jedan vrlo efikasan model za prikaz šahovske ploče, koji zadovoljava navedene uvjete, je model bitploča (eng. *bitboards*), koji koristi 64–bitne brojeve u kojem svaki bit predstavlja binarno stanje za jedno šahovsko polje.

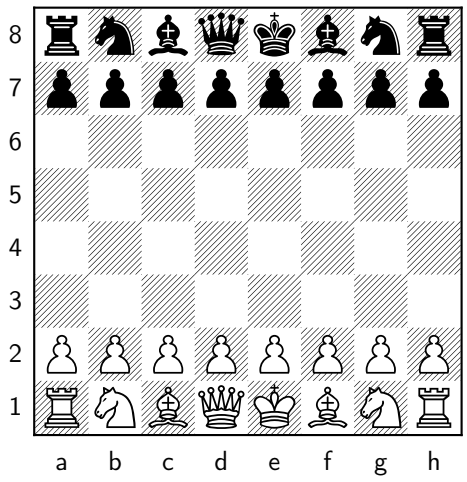
Cilj ovog rada je opisati teoretske principe iza rada šahovskih programa te stvoriti programsko rješenje koje pomoću tih principa za danu šahovsku poziciju procjenjuje optimalan potez (za danu dubinu pretrage stabla). Programsko rješenje temeljeno je na bitpločama te je zamišljeno na način da bude proširivo u budućnosti s novim tehnikama kako bi se programu dodatno poboljšale performanse, odnosno kvaliteta odabranih poteza. U tu svrhu, korišten je program izveden u sklopu diplomskog rada Šahovski računalni program[1], koji nudi grafičko sučelje za igranje šaha te generator legalnih poteza. Navedeni program pretvoren je u kompletni šahovski *engine* nadogradnjom funkcije statičke evaluacije pozicije te implementacijom *minimax* algoritma za pretragu stabla igre. Algoritam je zatim proširen i α–β rezanjem kako bi mu se poboljšala asimptotska kompleksnost.

U uvodnom poglavlju rada, ukratko se opisuje šah kao igru te sažima njena pravila. Naglašeni su posebni slučajevi i iznimke u pravilima. U drugom poglavlju promotrena je teoretska podloga i tehnike korištene pri radu programa koji igra šah. Konačno, u trećem poglavlju pokazani su i komentirani dobiveni rezultati te način korištenja programa za određivanje šahovskih poteza, koji je priložen uz rad.

2. PRAVILA ŠAHA

Šah je igra na ploči za dva igrača, crnog i bijelog. Akciju jednog igrača zovemo polupotez (eng. *ply*), a dva uzastopna polupoteza zovemo potez. Bijeli igrač povlači prvi polupotez, a zatim igrači povlače polupoteze naizmjence. Šah je igra s potpunom informacijom, odnosno oba igrača imaju potpuni pregled nad pozicijom i ne postoje nasumični faktori (kao npr. u igrama sreće).

2.1 Šahovska ploča



Slika 2.1. Početna šahovska pozicija

Šahovska ploča podijeljena je u 8x8 diskretnih polja, naizmjence obojanih crno i bijelo (počevši od donjeg lijevog ugla ploče). Vertikalni niz od 8 uzastopnih polja nazivamo stupcem (označeni slovima a-h, slijeva nadesno), horizontalni niz retkom (numerirani 1-8, od bijelog igrača prema crnom). Polja možemo jedinstveno označiti oznakom stupca i retka u kojem se nalaze (npr. polje e3). Uzastopni niz polja koja se dotiču samo u svojim vrhovima (i leže na nekom zamišljenom pravcu) zovemo dijagonalom.

2.2 Figure

Oba igrača započinju igru s identičnim skupom (osim boje) od 16 figura: kraljicom, kraljem, dva lovca, dva topa, dva skakača i osam pješaka. U retku najbližem igraču poredani su top, skakač, lovac, kraljica (ili dama), kralj, lovac, skakač i top. Sljedeći redak u potpunosti je ispunjen pješacima. Svaki tip figure ima različit način kretanja te napada. Figura se na dano polje može pomaknuti ako i samo ako se na tom polju već ne nalazi druga figura iste boje (odnosno, polje je ili prazno, ili je na njemu protivnička figura). Ukoliko se neka figura može pomaknuti na polje na kojem se nalazi protivnička figura, kažemo da ju napada. Ukoliko se tamo zaista i pomakne, kažemo da ju je zarobila ili "pojela". Pojesti je moguće samo protivničke figure. Dodatno, figura ne može preskočiti druge figure da bi došla do svoje destinacije, osim u slučaju skakača. Situaciju u kojoj je napadnut kralj nazivamo šah.

Cilj igre je zarobiti protivničkog kralja, na način da ga dovedemo u poziciju u kojoj se on nalazi pod šahom i ne može iz njega izaći u sljedećem potezu. Tu situaciju nazivamo mat. Igru je moguće završiti na još jedan način, a to je u neriješenom scenariju koji se naziva pat. On nastaje u situaciji kad protivnik nema legalnih poteza, ali mu kralj nije u šahu, ili ponekad primjenom posebnih pravila (npr. pravilo pedeset poteza pri kojem mora doći do pobjede unutar pedeset poteza).

Osim opisanih, postoje i neki iznimni potezi poput *en passant*, rokade ili promocije pješaka. Promotrit ćemo pobliže pravila kretanja svake pojedine figure te spomenuti ove iznimne poteze kod figure za koje su vezani.

2.2.1 Top

U danom polupotezu, top se kreće samo unutar jednog retka ili jednog stupca. Moguće ga je pomaknuti za bilo koji broj polja. Top je vrlo jaka figura kad se u danom stupcu nalazi sam jer tad njegova pravila kretanja daju jaku kontrolu nad pločom.

2.2.2 Lovac

Lovac se kreće slično kao i top, ali unutar dijagonala, umjesto redaka i stupaca. Kako dijagonale uvijek sadrže polja iste boje, jedna posljedica ovog pravila je asimetrija između dva lovca koja dani igrač posjeduje: lovac koji započne igru na bijelom polju neće nikad moći doći na crno polje i obrnuto. Zbog toga se u šahu par lovaca koji se nalaze na poljima suprotne boje smatra vrijednijim od zbroja vrijednosti dva lovca.

2.2.3 Kraljica

Kraljica se u danom polupotezu može kretati ili kao top, ili kao lovac. Zbog toga se obično smatra najjačom figurom.

2.2.4 Skakač

Skakač se u danom polupotezu može pomaknuti do bilo kojeg polja koje je od početnog udaljeno za dva polja u jednom smjeru (horizontalno ili vertikalno) i jedno polje u ortogonalnom smjeru. Njegovo kretanje možemo vizualizirati kao slovo L. Dodatno, skakač do svoje destinacije može doći neovisno o drugim figurama koje potencijalno stoje između početnog i krajnjen polja poteza.

2.2.5 Kralj

Kralj se kreće slično kao i kraljica, s ograničenjem da se u danom polupotezu može pomaknuti isključivo za jedno polje. Kralja je nemoguće pojesti. Ukoliko je kralj napadnut, njegov igrač prisiljen je u svojem sljedećem polupotezu taj napad obraniti, bilo pomicanjem kralja na neko drugo nenapadnuto polje ili skrivanjem kralja od napada pomicanjem neke druge figure. Situaciju u kojoj je napadnutog kralja nemoguće obraniti u danom polupotezu nazivamo šah mat i ona označava gubitak partije za vlasnika kralja.

Uz kralja i topa vežemo poseban potez zvan rokada. Ukoliko ni kralj, ni top još nisu pomaknuti, a polja između njih su prazna i nenapadnuta, kralja je moguće pomaknuti do topa, a zatim topa premjestiti na drugu stranu kralja. Rokadu s bližim topom nazivamo mala, a s daljim topom velika rokada.[2] Rokadu se obično smatra vrlo poželjnim potezom jer otvara igru omogućujući lakše kretanje topa, povećavajući usput sigurnost kralja.

2.2.6 Pješak

Naoko najjednostavnija figura čija su pravila kretanja najteža za opisati zbog velikog broja iznimki. Pješak se može kretati samo naprijed (gledano relativno, iz pogleda igrača vlasnika) i to samo za jedno polje. Iznimno, ako dani pješak još nije pomaknut u trenutnoj partiji, može se pomaknuti i za dva polja unaprijed.

Pješak je jedina figura koja jede druge figure pokretom drukčijim od svojeg uobičajenog kretanja: pješak može jesti jedino pomicanjem za jedno polje dijagonalno i to opet samo u smjeru primicanja protivničkoj strani. Tu je bitno napomenuti i poseban potez *en passant* (franc. "u pokretu"): ukoliko se pješak pomakne za dva polja unaprijed (jer mu je to prvi pomak) te se time postavi uz bok protivničkom pješaku, protivnički pješak ga ima pravo pojesti kao da se zapravo pomaknuo za samo jedno polje.[3]

Ukoliko pješak dosegne rubno polje na protivničkoj strani, moguće ga je pretvoriti u bilo koju drugu figuru osim kralja što zovemo promocija pješaka.

3. TEORIJSKI PRINCIPI RADA ŠAHOVSKIH PROGRAMA

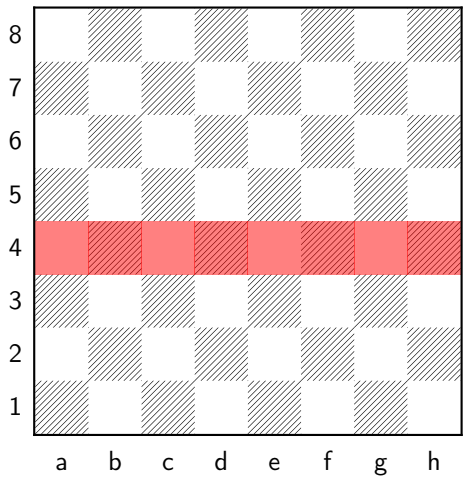
3.1 *Bitboard* metoda

Kao što je spomenuto u uvodu, *bitboard* je oblik prikaza informacije na šahovskoj ploči u obliku 64-bitnog broja. Svaki bit odgovara jednom polju (po nekoj unaprijed uspostavljenoj konvenciji, npr. nulti bit može odgovarati polju a1, prvi polju a2, itd.), a njegova vrijednost prikazuje neku binarnu informaciju (primjerice, "polje je u napadu", "na polju stoji lovac", "na polju se nalazi crna figura"). Iz ovog je jasno da je za potpuni opis danog stanja šahovske ploče potrebno imati minimalno 8 *bitboarda* (po jedan za svaki tip figure i dva za razlikovanje bijelih od crnih figura). Osim ovih, u samom izračunu koristit ćemo i mnoge pomoćne *bitboarde*, poput reprezentacije napadnutih polja ili *bitboarda* vezanog za pojedini redak ili stupac.

Kako moderni procesori većinom koriste 64-bitne registre, na mnogim procesorskim arhitekturama moguće je spremiti jedan čitav *bitboard* u samo jedan procesorski registar. To ovu reprezentaciju ploče čini vrlo kompaktnom. Štoviše, budući da bitnim (eng. *bitwise* operacijama nad *bitboardima* možemo modelirati logičke operacije nad rasporedom figura na ploči, ovom metodom možemo postići vrlo visoke performanse jer su takve operacije u procesorima izuzetno optimizirane. Kombinacijom nekoliko elementarnih *bitwise* operacija možemo postići efikasno generiranje legalnih poteza za sve vrste figura, kao i razne posebne provjere konfiguracije ploče, poput provjere je li top sam u stupcu. Drugim riječima, *bitboard* metoda omogućava nam da korištenjem *bitwise* operacija nad *bitboardima* modeliramo *Boolean* algebru nad izrazima vezanim uz konfiguraciju šahovske ploče.

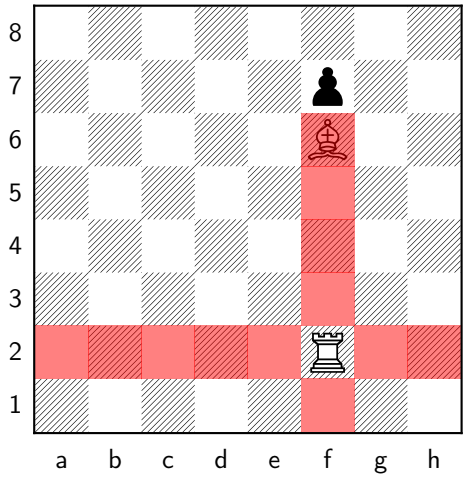
Iz perspektive *bitboard* metode, šahovske figure možemo podijeliti u dvije bitne skupine koje je potrebno tretirati na drukčiji način: skupina neklizajućih (eng. *non–sliding*, kralj, skakač i pješak) i klizajućih (eng. *sliding*, top, kraljica, lovac) figura. Ove dvije skupine razlikuju se po tom što su napadnuta polja neklizajućih figura određena samo poljem na kojem se figura nalazi, dok napadnuta polja klizajućih figura ovise i o drugim figurama jer one mogu blokirati njihove linije napada. Primjerice, na slici 3.2, top na f2 ne napada pješaka na f7 jer liniju napada blokira lovac na f6 te ju na tom polju i prekida. Zbog ovog su algoritmi za generiranje poteza neklizajućih figura jednostavniji.

Ujedno je bitno napomenuti kako se iste procedure mogu koristiti za generiranje poteza danog tipa figure neovisno o boji. Sve što trebamo napraviti je predati proceduri *bitboard* koji predstavlja poziciju figura nekog tipa za igrača (bijelog ili crnog) za kojeg želimo generirati poteze.



Slika 3.1. *Bitboard* 4. retka

Na priloženim slikama možemo vidjeti grafički prikaz jednostavnih *bitboarda* ukoliko zamislimo da su polja na kojima su bitovi s vrijednošću 1 označena crvenom bojom. Slika 3.1 predstavlja *bitboard* četvrtog retka, dok slika 3.2 predstavlja napade bijelog topa na slici. Promotrit ćemo nad par primjera kako se ovakvi *bitboardi* mogu kombinirati u odgovore na pitanja o konfiguraciji ploče.



Slika 3.2. Prekid linije napada topa drugom figurom

3.1.1 Napadnuti bijeli pješaci

Primjerice, može nas zanimati koliko je bijelih pješaka trenutno napadnuto. To pitanje možemo raščlaniti na dva pitanja:

1. Gdje se nalaze bijeli pješaci?
2. Koja polja su trenutno napadnuta od strane crnih figura?

U pseudokodu 3.1 možemo vidjeti primjer izračuna *bitboarda* napadnutih bijelih pješaka ukoliko pretpostavimo da smo već implementirali funkcije koje nam vraćaju *bitboarde* za bijele figure, pješake te za napadnute figure. Ukoliko bismo dodatno htjeli znati koliko je napadnutih bijelih pješaka, sve što trebamo je prebrojati broj postavljenih bitova u rezultatu. Kao što vidimo, metoda je vrlo intuitivna i lako se proširuje na izračun odgovora na vrlo kompleksna pitanja o konfiguraciji ploče.

```
bijeli_pjesaci = bitboard_bijelih() & bitboard_pjesaka();
napadnuta = bitboard_napadnutih_polja();
napadnuti_bijeli_pjesaci = bijeli_pjesaci & napadnuta;
```

Izvorni kod 3.1. Pseudokod izračuna napadnutih bijelih pješaka

3.1.2 Top u otvorenom stupcu

Nadalje, može nas zanimati je li dani top u otvorenom stupcu, odnosno u stupcu u kojem nema drugih figura. To pitanje je zanimljivo jer takvu poziciju topa preferiramo budući da vodi do boljih poteza.

U izvornom kodu 3.2 možemo vidjeti funkciju za ovaj izračun koja je korištena u programskom rješenju ovog rada. Varijablu *tempRooks* inicijaliziramo s *bitboardom* svih topova na ploči. Za svakog topa, računamo *bitboard* svih zauzetih polja u njegovom stupcu pomoću *bitwise AND* operacije, postavljamo bit koji predstavlja samog topa na 0, a zatim prebrojavamo koliko je bitova postavljeno. Ukoliko je taj broj 0, top je sam u stupcu. Za dodatnu provjeru je li top u poluotvorenom stupcu (stupcu u kojem se nalaze samo protivničke figure), za svakog topa dodatno pronalazimo *bitboard* protivničkih figura. U provjeri za poluotvoreni stupac onda i te bitove postavljamo na 0.

```
public static int evaluateRooksOnOpenRanks(Board board) {
  ...
  UInt64 tempRooks = board.whiteRooks | board.blackRooks;
  UInt64 enemyPieces;
  ...
  while (tempRooks != 0) {
    uint pos = BitOps.firstOne(tempRooks);
```

```
    if ((G.BITSET[pos] & board.whitePieces) != 0) {
        // bijeli top
        enemyPieces = board.blackRooks;
        ...
    } else {
        // crni top
        enemyPieces = board.whiteRooks;
        ...
    }

    // zauzeta polja u stupcu
    UInt64 occupiedFileSquares = board.occupiedSquares &
  G.FULLFILEMASK[pos];

    // je li stupac prazan (osim topa)
    bool openFile = BitOps.bitCnt(occupiedFileSquares ^
  G.BITSET[pos]) == 0;

    // jesu li u stupcu samo top i protivnikove figure
    UInt64 enemyPiecesInFile = enemyPieces & G.FULLFILEMASK[pos];
    bool semiOpenFile = BitOps.bitCnt(occupiedFileSquares ^
  enemyPiecesInFile) == 0;

    if (openFile) {
        // top je u otvorenom stupcu
    } else if (semiOpenFile) {
        // top je u poluotvorenom stupcu
    }

    tempRooks ^= G.BITSET[pos];
  }
  ...
}
```

Izvorni kod 3.2. Kod koji provjerava je li top u otvorenom stupcu

3.2 Funkcija pretrage

Pod pojmom funkcije pretrage smatramo algoritam koji, krenuvši od neke početne pozicije, pretražuje stablo igre, u nastojanju da pronađe optimalan potez za trenutnog igrača. Budući da je šah igra potpune informacije, odnosno nema tajnih ili nasumičnih elemenata, stablo igre šaha moguće je u teoriji (ali ne i u praksi) u potpunosti pretražiti. Ujedno, šah je igra nulte sume, što znači da ne postoje odvojeni pojmovi vlastitog dobitka i protivnikovih gubitaka, nego su to suprotni pojmovi. U igrama nulte sume s dva igrača, dobitak jedne strane iznosi točno negativni iznos gubitka druge strane, odnosno vrijedi $D = -G$ i $D + G = 0$, ukoliko D predstavlja dobitak jednog igrača, a G gubitak drugog.[4, str. 216]

3.2.1 Minimax

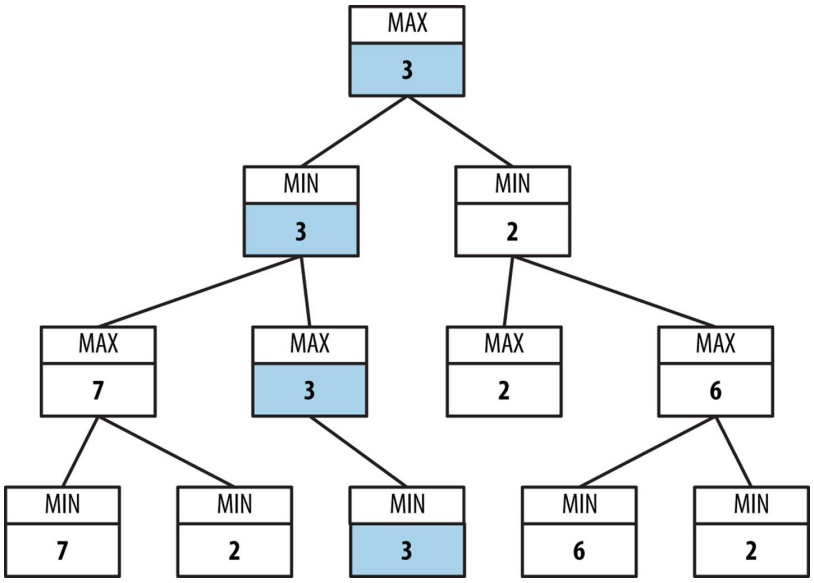
Jedan od najčešće korištenih algoritama za odabir optimalnog poteza je *minimax*, koji je formuliran u okviru teorije igara. Osnovna ideja *minimax* algoritma je minimiziranje maksimalnog gubitka za trenutnog igrača, odnosno odabir onog niza poteza koji će dovesti do najmanjeg mogućeg gubitka, čak i ako drugi igrač igra savršeno.[5, str. 302] Uz *minimax* vežemo i pojam *maximin* algoritma, koji je prirodni obrat, odnosno algoritam koji maksimizira minimalnu moguću dobit. U igrama nulte sume, u kojima je dobitak jednog igrača gubitak drugog, ova dva algoritma su ekvivalentna. Vrijednost koju *minimax* računa zovemo *minimax vrijednost*. Sljedeći izraz predstavlja formalnu definiciju *maximin* vrijednosti, koja je u igrama nulte sume jednaka *minimax* vrijednosti. U izrazu, $\underline{v_i}$ predstavlja *maximin* vrijednosti, v_i predstavlja funkciju dobitka za i-tog igrača, a_i potez i-tog igrača, a a_{-i} poteze svih igrača osim i-tog:

$$\underline{v_i} = \max_{a_i} \min_{a_{-i}} v_i(a_i, a_{-i}) \tag{3.1}$$

Drugim riječima, *maximin* vrijednost predstavlja najveći mogući dobitak ako pretpostavimo odabir najbolje mogućeg poteza za trenutnog igrača, a pritom pretpostavimo da će ostali igrači odabrati najgore moguće poteze za njega.

Budući da šah ima samo tri ishoda (pobjedu, gubitak i pat), dobitak (odnosno gubitak) bismo mogli opisati pomoću samo tri vrijednosti, primjerice 1, 0 i -1. U tom slučaju, *minimax* algoritam teoretski bi mogao pretražiti partiju šaha do samog kraja te kao *minimax* vrijednost vratiti jednu od te tri vrijednosti. Svaka bi se šahovska pozicija onda mogla opisati kao dobitna, gubitna ili pozicija koja vodi do neriješenog rezultata.[6] Takva rješenja postoje za jednostavnije igre kao što je primjerice *Križić–kružić*. Međutim, zbog prije spomenutih praktičnih ograničenja, šah nije moguće na taj način riješiti. Taj problem zaobilazimo ograničavanjem dubine pretrage stabla, no na taj način najčešće nećemo doći do završnih pozicija (mat ili pat) pa im ne možemo na direktan način pridružiti dobitak, odnosno gubitak. Zbog toga vrijednost nezavršnih pozicija dobijamo heuristički, korištenjem

statičke funkcije evaluacije, koja procjenjuje dobitak dane pozicije.[7]



Slika 3.3. Vizualizacija *minimax* algoritma

Na slici 3.3 možemo vidjeti vizualizaciju *minimax* algoritma nad stablom neke izmišljene partije. Primjećujemo da se na čvorovima izmjenjuju koraci maksimizacije i minimizacije, predstavljajući naizmjence poteze prvog i drugog igrača. Vrijednosti pozicija na listovima stabla izračunati su pomoću funkcije za statičku evaluaciju pozicije. Te vrijednosti se zatim propagiraju prema gore, gdje prethodni čvor uzima najmanju ili najveću moguću vrijednost svoje djece, i tako sve do korijena stabla. Plavo je označen najbolji niz poteza za igrača koji je na redu u trenutku pozicije koju korijen stabla predstavlja. Takav niz poteza zovemo *glavna varijacija* (eng. *principle variation*). Također, možemo primjetiti da glavna varijacija ne završava u poziciji koja je najbolja za početnog igrača od svih mogućih pozicija, ali to jest najbolja pozicija u koju garantirano možemo doći, neovisno o tome koje poteze odigrava drugi igrač.

Iz vizualizacije je očigledno da algoritam ima eksponencijalnu vremensku kompleksnost: za svaku poziciju postoji B različitih mogućih poteza, gdje B u teoriji stabala zovemo faktor grananja. Drugim riječima, vremenska kompleksnost ovog algoritma je $\mathrm{O}(B^D)$, gdje je D dubina pretrage. U prosječnoj šahovskoj poziciji postoji oko 30 legalnih poteza[8][6], pa vrijedi $B \sim 30$. Moderna računala izvršavaju oko 3 GHz $\sim$ 3.000.000 operacija po sekundi pa ako pretpostavimo da jednu poziciju možemo generirati i evaluirati pomoću $\sim$ 60.000 instrukcija, to nas dovodi do oko 50.000 pozicija po sekundi.[8].

U tablici 3.1 možemo vidjeti rezultat takvog izračuna za nekoliko dubina pretrage. Iz tablice je lako zaključiti da je obični *minimax* algoritam nepraktičan za šah za dubine veće od 5-6 na današnjim osobnim računalima, što opravdava nužnost dodatnih tehnika koje ćemo opisati u narednim sekcijama.

Konačno, promotrimo ćemo pseudokod *minimax* algoritma:

Dubina (polupoteza)	Broj pozicija	Vrijeme pretrage
2	900	18 ms
3	27.000	540 ms
4	810.000	16.2 s
5	24.300.000	8 minuta
6	729.000.000	4 sata
7	21.870.000.000	5 dana

Tablica 3.1. Procjena broja pozicija i vremena pretrage za šahovsko stablo po dubini pretrage

```
def maximize(pozicija, dubina):
    potezi := legalni_potezi(pozicija)

    if dubina = 0 or length(potezi) = 0:
        return evaluiraj(potezi)

    najveci_dobitak = −∞

    for potez in potezi:
        pozicija := ucini_potez(pozicija, potez)
            d := minimize(pozicija, dubina - 1)
            najveci_dobitak := max(najveci_dobitak, d)
        pozicija := povuci_potezi(pozicija, potez)

    return najveci_dobitak

def minimize(pozicija, dubina)
    potezi := legalni_potezi(pozicija)

    if dubina = 0 or length(potezi) = 0:
        return evaluiraj(potezi)

    najmanji_dobitak = +∞

    for potez in potezi:
        pozicija := ucini_potez(pozicija, potez)
            d := maximize(pozicija, dubina - 1)
            najveci_dobitak := min(najmanji_dobitak, d)
        pozicija := povuci_potezi(pozicija, potez)
```

```
31      return najmanji_dobitak
```

Izvorni kod 3.3. Pseudokod *minimax* algoritma

Iz priloženog pseudokoda 3.3 možemo vidjeti da se *minimize* i *maximize* koraci međusobno rekurzivno pozivaju, predstavljajući naizmjence minimizirajućeg te maksimizirajućeg igrača. Igrači evaluiraju sve dozvoljene poteze te biraju onaj koji minimizira (odnosno maksimizira) garantirani dobitak, uzimajući u obzir *dubina* polupoteza unaprijed. Kad dođemo na dubinu 0, ili ukoliko naiđemo na završnu poziciju (gdje nema legalnih poteza), evaluiramo poziciju pomoću statičke funkcije evaluacije te vraćamo rezultat.

U praktičnim implementacijama *minimax* i sličnih algoritama, potrebno je u prvom pozivu algoritma, umjesto dobiti, vratiti potez za koji tu dobit vežemo. Zbog toga je za prvi poziv potrebno implementirati funkciju koja je vrlo slična opisanom pseudokodu, ali uz najbolji rezultat pamti i potez koji ga je proizveo te vraća pozivatelju sam potez. Ta promjena nema bitne konceptualne značajke te je samo implementacijski detalj.

3.2.2 Negamax

U igrama nulte sume gore opisani algoritam možemo svesti na jednostavniju formu koja koristi samo korak maksimizacije, ali je u drugim aspektima ekvivalentna. Takav algoritam se ne razlikuje ni u performansama od *minimaxa*, ali je često jednostavniji za implementirati budući da nisu potrebne dvije zasebne procedure (jedna za maksimizaciju, a druga za minimizaciju). Takav algoritam naziva se *negamax* i često je korišten u šahovskim programima.

Algoritam se oslanja na činjenicu da je u igrama nulte sume najbolji potez trenutnog igrača ujedno i najlošiji potez za protivnika, odnosno

$$max(a, b) = -min(-a, -b) \tag{3.2}$$

gdje a i b predstavljaju vrijednosti neka dva poteza. Koristeći tu jednakost, možemo transformirati rezultate dobivene za minimizirajućeg igrača, te efektivno dobiti algoritam koji koristi samo jednu funkciju kao u pseudokodu 3.4.[5, str. 311]

```
1  def negamax(pozicija, dubina, boja):
       potezi := legalni_potezi(pozicija)
3
       if dubina = 0 or length(potezi) = 0:
5          return boja * evaluiraj(potezi)

7      najveci_dobitak = −∞

9      for potez in potezi:
```

```
        pozicija := ucini_potez(pozicija, potez)
11          d := -negamax(pozicija, dubina - 1, -boja)
            najveci_dobitak := max(najveci_dobitak, d)
13      pozicija := povuci_potezi(pozicija, potez)

15  return najveci_dobitak
```

Izvorni kod 3.4. Pseudokod *negamax* algoritma

U navedenom pseudokodu treba primjetiti novi parametar *boja*, koji postavljamo na 1 za maksimizirajućeg, odnosno -1 za minimizirajućeg igrača i koji invertiramo u svakom novom rekurzivnom pozivu. Navedeni parametar koristi se za prilagodbu rezultata koji vraća statička funkcija evaluacije budući da ona vraća rezultate koji su apsolutni (veći rezultati za maksimizirajućeg, manji rezultati za minimizirajućeg igrača), a *negamax* očekuje rezultate relativne za trenutnog igrača. Ujedno, rezultat funkcije *negamax* negiramo, iskorištavajući pravilo iz jednadžbe (3.2).

3.2.3 α–β rezanje

α–β rezanje (eng. α–β *pruning*) je tehnika pomoću koje možemo značajno reducirati stablo stanja koje moramo pretražiti da bismo dobili *minimax* rezultat. Rezultat dobiven α–β rezanjem uvijek je ekvivalentan *minimax* rezultatu, ali s, u prosječnom slučaju, uvelike reduciranim eksponentom vezanim za dubinu pretrage. Kada koristimo α–β rezanje, vremenska kompleksnost *negamax* algoritma smanjuje se s $\mathrm{O}\big(B^D\big)$ na $\mathrm{O}\big(B^{D/2}\big)$ u najboljem te $\mathrm{O}\big(B^{\frac{3}{4}D}\big)$ u prosječnom slučaju, gdje je D dubina pretrage, a B broj mogućih poteza u danoj poziciji.[5, str. 322] Broj poteza koje treba evaluirati u danoj poziciji smanjuje se s 30 do 40 na 7 do 8.[8] To omogućuje gotovo udvostručenje dubine, odnosno broja polupoteza koje je moguće pretražiti za isto vrijeme.

Osnovna ideja α–β algoritma je korištenje informacija koje su već dostupne tijekom izvođenja *negamaxa* kako bismo odrezali (prestali evaluirati) one pozicije za koje već znamo da nikako ne mogu utjecati na trenutnu granu. Algoritam tijekom izvođenja dodatno pamti dvije vrijednosti:

α — najbolja dosad ostvarena dobit za trenutnog igrača

β — najbolja dosad ostvarena dobit protivničkog igrača

Ukoliko na danoj dubini pronađemo poziciju koja je za trenutnog igrača ocijenjena kao bolja nego što je neka prethodno pronađena pozicija ocijenjena za protivničkog igrača, odnosno ukoliko $dobit \geqslant \beta$, možemo prekinuti evaluaciju te pozicije. To vrijedi zbog toga što bi protivnički igrač u prošloj rundi svakako mogao odabrati poziciju koja njemu donosi dobit β pa nikako ne bismo mogli završiti u trenutnoj grani.

Vrijednosti α i β ponašaju se kao granične vrijednosti unutar kojih se mora nalaziti dobit trenutne grane kako bismo ju uzeli u obzir. Interval $[\alpha, \beta]$ zbog toga još nazivamo i α–β *prozor*. Ukoliko procijenjena dobit trenutne grane padne izvan α–β prozora, iz trenutnog *negamax* poziva to signaliziramo vraćanjem vrijednosti α ili β, umjesto pravog rezultata grane, ovisno o tome je li prozor probijen prema dolje ili gore. Dodatno, zbog istih razloga kao i u *negamax* algoritmu, *alpha* i *beta* vrijednosti u svakom sljedećem pozivu se negiraju i zamjenjuju mjesta (jer je najbolji potez trenutnog igrača najgori potez protivničkog i obrnuto). Navedeno se može vidjeti u pseudokodu 3.5.

```
def alphabeta(pozicija, dubina, alpha, beta, boja):
    potezi := legalni_potezi(pozicija)

    if dubina = 0 or length(potezi) = 0:
        return boja * evaluiraj(potezi)

    najveci_dobitak = −∞

    for potez in potezi:
        pozicija := ucini_potez(pozicija, potez)
            d := -alphabeta(pozicija, dubina - 1, -beta, -alpha,
  -boja)
            najveci_dobitak := max(najveci_dobitak, d)
        pozicija := povuci_potezi(pozicija, potez)

        if d >= beta:
          return beta

        alpha = max(alpha, d)

    return alpha
```

Izvorni kod 3.5. Pseudokod α–β algoritma

Tijekom evaluacije stabla igre, α–β prozor se sužava kako dobijamo nove informacije, a time se i povećava vjerojatnost da ćemo odrezati granu koju trenutno pretražujemo. Iz tog razloga nam je izuzetno bitno da poteze evaluiramo od najboljeg prema najlošijem jer time povećavamo udio grana koje možemo odrezati. Naravno, nemoguće je znati koji potez je najbolji (jer upravo zbog toga i izvršavamo algoritam), ali je moguće heuristikama poredati poteze približno dobro kako bismo ostvarili što veće ubrzanje. Budući da je ušteda u vremenu eksponencijalna, isplati se relativno mnogo vremena provesti u sortiranju poteza prije nego se započne izvršavanje samog α–β algoritma. Jedna očita taktika, koja je korištena i u

programskom rješenju ovog rada, je sortiranje legalnih poteza prema rezultatu koji vraća funkcija za statičku evaluaciju. U praksi je pokazano da se za šah u većini slučajeva poteze može sortirati dovoljno dobro da bi se ostvarila vremenska kompleksnost vrlo blizu onoj koja je ostvariva u najboljem slučaju.[7, str. 43]

Algoritam nije trivijalno paralelizirati jer naivni način paralelizacije (po jedan *thread* po grani stabla) nije moguć budući da se algoritam oslanja na rezultat iz jedne grane da bi ostvario rezanje drugih. Ipak, moguće je stablo podijeliti na način da se prva grana do kraja pretraži sekvencijalno i time inicijalizira α–β prozor, a zatim se druge grane na svakoj dubini pretraže paralelno. Ukoliko su potezi dobro sortirani, većinom će prvi pretraženi potez biti i najbolji potez pa će ostale grane moći biti odrezane bez daljnjeg podešavanja α–β prozora.[9]

3.2.4 Drugi algoritmi pretrage

Osim spomenutih, postoje i drugi algoritmi koji na razne načine dodatno pokušavaju ubrzati pretragu. Jedan od prvih takvih algoritama je SSS*, modeliran prema poznatom A* algoritmu. SSS* ne računa samo jednu vrijednost nego razvija optimalnu strategiju za jednog igrača postepenim rezanjem stabla igre dok na svim maksimizirajućim čvorovima stabla ne ostane samo jedan potez. Takvo stablo predstavlja optimalnu strategiju za maksimizirajućeg igrača.[10]. Kasnije se ispostavilo da u praksi nije bolji nego α–β rezanje. Nešto kasnije razvijen je algoritam NegaScout koji pretpostavlja da je prvi pretraženi potez u glavnoj varijaciji, a zatim koristi nulti α–β prozor (takav da $\alpha = \beta + 1$) kako bi to provjerio. Što je prozor manji, to će se događati više rezanja i algoritam će se izvršiti brže. Međutim, ako se ispostavi da je pretpostavka netočna, NegaScout će morati ponovno pretražiti potez s normalnim prozorom. To može dovesti do toga da bude sporiji nego obično α–β rezanje u slučajevima kad potezi nisu dobro sortirani.

Kasnije je pokazano da je SSS* ekvivalentan korištenju α–β algoritma u kombinaciji s transpozicijskim tablicama.[11] Ista grupa autora razvila je i novi MTD-f algoritam, koji isključivo radi pozive s nultim prozor i postepeno konvergira prema točnom rješenju.[12] MTD-f pamti pretražene pozicije pomoću transpozicijskih tablica kako bi ubrzao ovaj proces. MTD-f se koristi u mnogim modernim šahovskim programima i često je najefikasniji od navedenih algoritama.

3.2.5 Heuristike

Osim modificiranja samog algoritma pretrage, moguće je implementirati i razne heuristike koje ubrzavaju pretragu ili povećavaju kvalitetu pretrage. Spomenut ćemo neke od tih heuristika.

Quiescence pretraga

Kod pretrage stabla igre se javlja tzv. *problem horizonta*. Ukoliko stablo pretražujemo s konstantnom dubinom, događat će se da ćemo na kraju pretrage pronaći potez koji se čini kao da je dobar, no već u sljedećem polupotezu (ili njih nekoliko) dovodi do velikog gubitka (npr. gubitak kraljice). To možemo izbjeći primjenom *quiescence* (hrv. stanje tišine) pretragom, koja produbljuje pretragu u slučajevima kad pozicija na kojoj bismo stali s pretragom nije strateški "tiha". Pretragu nastavljamo sve dok ne dođemo do tihe pozicije. Najčešće se to odnosi na pozicije u kojima nema poteza koji jedu figure, ali može se odnositi i na pozicije u kojima kralj nije u šahu i slično.[13]

Transpozicijske tablice

Transpozicijske tablice su struktura podataka u koju algoritam pretrage zabilježava rezultate pretrage za danu poziciju. Najčešće su implementirani kao *hash* tablice u kojima se kao ključ koristi *hash* dane šahovske pozicije. [7, str. 83][8] U tablicu se sprema rezultat za cijelo podstablo pa, ukoliko ponovno naiđemo na istu poziciju, možemo pretragu tog podstabla zamijeniti pristupom *hash* tablici, što je mnogo brža operacija. Budući da tijekom jedne igre bude pronađeno previše šahovskih pozicija da bi se sve mogle spremiti, stare pozicije kojima se dugo nije pristupilo se iz tablice izbacuju.

Postupno produbljivanje (eng. *iterative deepening*)

Tehnika se sastoji od pretraživanja stabla s postupnim povećavanjem dubine pretrage, odnosno prvo obavljamo pretragu s dubinom 1, zatim s dubinom 2, zatim s dubinom 3, i tako dalje, do isteka vremena. To donosi nekoliko prednosti, uključujući priliku za bolje sortiranje poteza kako bismo omogućili α–β algoritmu da veći dio stabla odreže, kao i interaktivnost prema korisniku jer smo u svakom trenutku u mogućnosti dati dobru procjenu poteza (odnosno, najbolju raspoloživu u tom trenutku). Ova tehnika se uvelike oslanja na transpozicijske tablice jer bi bez njih bila prespora budući da u svakom produbljivanju na razinu N ispočetka pretražujemo sve razine $< N$. Korištenjem transpozicijskih tablica smanjuje to usporenje i čini ga zanemarivim.

Aspiracijski prozori

Tehnika koja se oslanja na ideju sličnu ranije spomenutom nultom prozoru prilikom NegaScout pretrage, ali u kojoj koristimo prozor širine veće od 1 (odnosno, $alpha - beta > 1$). Budući da uži prozori omogućavaju više rezanja, a time i bržu pretragu, to može dovesti do boljih performansi algoritma.[8][14] Kako bi takav algoritam dobro funkcionirao, moramo napraviti dobru početnu pretpostavku oko širine i pozicije prozora. Ukoliko je pretpostavka loša, morat ćemo napraviti još najmanje jednu dodatnu pretragu istog poteza pa to može dovesti i do usporenja.

Nulti potez (eng. *null–move*)

Oslanja se na ideju da je u većini pozicija igraču bolje odigrati potez nego ne odigrati ništa. Iako neodigravanje poteza u šahu nije dozvoljeno, šahovski programi mogu spekulativno simulirati situaciju u kojoj igrač nije napravio nikakav potez (nego ga se efektivno preskočilo). Ukoliko je njegova pozicija i dalje jaka te proizvodi β rez u α–β algoritmu, možemo pretpostaviti da bi se rez gotovo sigurno dogodio i da smo povukli potez. U tom slučaju granu možemo odsjeći ranije budući da svi potezi probijaju α–β prozor. Međutim, potrebno je paziti na tzv. *zugzwang* (hrv. *prisiljen da se pomakne*) situacije gdje bi igraču zaista bilo bolje da ne povuče potez jer su mu svi legalni potezi loši. U toj situaciji heuristika nultog poteza može napraviti pogrešku.[8][15]

Potez ubojica (eng. *killer move*)

Ukoliko je neki potez prouzročio rezanje tijekom α–β pretrage, isplati ga se zapamtiti, skupa s dubinom stabla na kojoj se to dogodilo. Ukoliko se dalje tijekom pretrage isti potez ponovi i u nekoj drugoj grani na sličnoj poziciji, velika je vjerojatnost da će to ponovno biti najbolji potez.[8] Heuristika poteza ubojice pamti takve poteze i pretražuje ih prve, ukoliko su gornji uvjeti zadovoljeni.

3.3 Funkcija evaluacije

Funkcija evaluacije služi nam za procjenu vrijednosti dane pozicije, što nam omogućava izvođenje algoritama pretrage bez evaluacije stabla igre skroz do njegovih listova, odnosno do završnih pozicija igre.[6][5, str. 306]

Takva funkcija za šah je nužno heuristična jer prava vrijednost pozicije ovisi o tome može li se ili ne iz takve pozicije pobijediti, što bi zahtjevalo da stablo igre pretražimo do kraja. Zato za funkciju evaluacije kažemo da je *statična*, pod čim podrazumijevamo da gleda samo trenutnu konfiguraciju ploče te na temelju iste pokušava procijeniti koji od igrača je u prednosti te tu prednost kvantificirati.

Funkcija evaluacije vraća numeričku vrijednost, i to 0 ukoliko procjenjuje da je pozicija izjednačena za oba igrača, a pozitivan ili negativan broj ukoliko u poziciji prednost ima maksimizirajući, odnosno minimizirajući igrač. Za rezultat funkcije uobičajeno je korištenje *centipawns* mjere. Ime te jedinice odnosi se na 1/100 vrijednosti pješaka pa u tom sustavu mjerenja pješak ima vrijednost 100, a sve ostale vrijednosti skalirane su relativno toj veličini. Ova mjera je značajna jer omogućuje precizniju usporedbu pozicija te preciznije relativno skaliranje značaja pojedinih figura, poteza i konfiguracije ploče, te je stoga vrlo rasprostranjena.

Funkciju evaluacije moguće je napraviti vrlo kompleksnom i u nju ugraditi mnogo strateškog znanja, no postoji prirodan sukob između sofisticiranosti funkcije i njenih performansi.

Budući da se funkcija poziva za svaku pretraženu poziciju, poželjno je da je veoma brza. Dodatno, postoji nekoliko eksperimentalnih rezultata koji navode na zaključak da su vrlo kompleksne funkcije evaluacije nepoželjne. Primjerice, utvrđeno je da programi koji pretražuju samo 1 polupotez dublje pobjeđuju u 80% slučajeva.[16] Štoviše, jačina igranja se povećava linearno s dubinom pretrage. To povećanje prestaje nakon neke dubine, no trenutni šahovski programi nisu blizu te granice pa se u većini slučajeva isplati povećati dubina pretrage.

Promotrit ćemo neke od strategija korištenih prilikom procjene šahovskih pozicija.

Procjena materijala

Procjena materijala odnosi se na jednostavno pridruživanje vrijednosti svakom tipu figure te zbrajanje vrijednosti za sve figure na ploči. Pritom se vrijednosti figura za jednog igrača uzimaju kao pozitivne, a za drugog kao negativne, tako da, ukoliko je materijal izjednačen, procjena materijala iznosi 0. Ponekad se za neke kombinacije figura daje dodatni bonus, kao npr. za par lovaca koji se smatra jakom kombinacijom.

Procjena mobilnosti

Procjenom mobilnosti pokušavamo kvantificirati koliko figure danog igrača imaju raspoloživih poteza. Ovakva provjera ima smisla jer što je manje mogućih poteza, to je vjerojatnije da će igrač biti prisiljen u za njega nepovoljnu poziciju. Slično kao i za procjenu materijala, broj poteza za sve figure jednog igrača se pribraja, a broj poteza za sve figure drugog igrača se od tog broja oduzima.

Procjena rasporeda figura

Procjena rasporeda figura je najširi pojam od navedenih. Odnosi se općenito na procjenu kvalitete rasporeda figura na ploči te koliko će vjerojatno takav raspored dovesti do pobjede igrača.

Jedna relativno jednostavna, a visoko korisna tehnika su *piece–square* tablice. Sastoji se od niza tablica, barem jedna za svaki tip figure, unutar kojih za svako polje na ploči zapisujemo vrijednost koju pribrajamo ukoliko figura tog tipa stoji na tom polju. Korištenjem takvih tablica u program možemo dodati mnogo ljudskog strateškog znanja o šahu. Primjerice, tablicama možemo penalizirati kraljicu koja stoji na rubu ploče, a pogotovo u kutevima, jer tako odbacuje veliki dio svoje napadačke moći. *Piece–square* tablice vrlo su korisne i za kralja, budući da se njegova završna igra razlikuje od igre u sredini partije. Pri završetku partije, tablicama možemo preferirati pozicije kralja u sredini ploče jer ga je tako teže matirati, dok na početku i u sredini partije želimo da je kralj zaštićen iza reda pješaka. U tom slučaju za kralja nam trebaju dvije različite tablice između kojih treba interpolirati kako igra napreduje. Zbog toga u sklopu procjene rasporeda figura, šahovski programi

procjenjuju i trenutnu fazu igre, primjerice pomoću broja figura koje su preostale na ploči.

Često se u sklopu procjene rasporeda figura procjenjuje i sigurnost kralja. To je moguće izvesti na razne načine, no neki od popularnih pristupa su provjera udaljenosti kralja od protivnikovih figura, provjera broja napadnutih polja oko kralja, provjera postoji li zaštitni red pješaka u blizi kralja i sl.

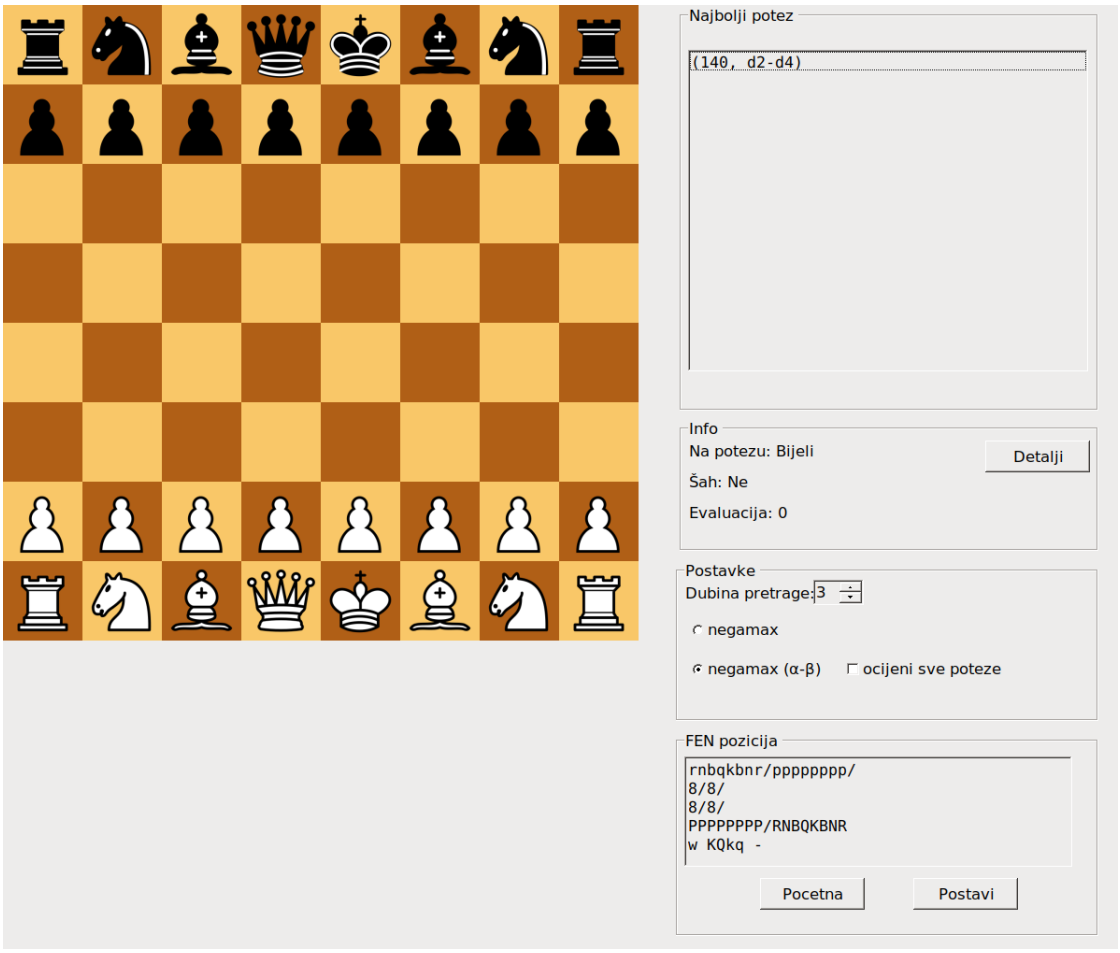
Posebna vrsta provjere rasporeda figura je provjera strukture pješaka. Struktura pješaka navodi ponašanje ostalih figura pa je bitno procijeniti jesu li dobro složeni. Primjer takve procjene je penaliziranje uduplanih, zaostalih i izoliranih pješaka. Uduplani pješaci su oni koji se nalaze u istom stupcu kao i neki drugi pješak iste boje. Zaostali pješaci su oni kojima je sljedeće polje napadnuto od strane protivničkog pješaka, a oni sami iza sebe nemaju pješaka koji ih može braniti. Izolirani pješaci su oni koji u stupcima pored sebe nemaju pješaka iste boje.

Konačno, postoji veći broj specifičnih situacija koje je moguće provjeravati, primjerice bonus za topove u otvorenim ili poluotvorenim stupcima, bonus za bijelog (crnog) topa u 7. (2.) retku, penal za zatvorenog lovca i sl.

4. O IMPLEMENTACIJI

Kao podloga za razvoj programskog rješenja za izračun poteza i procjenu najboljeg, korišteno je programsko rješenje izvedeno u sklopu diplomskog rada Šahovski računalni program[1]. Spomenuto rješenje nudi grafičko šahovsko sučelje te generator legalnih poteza.

Programsko rješenje u potpunosti je napisano u programskom jeziku C#. Razvijeno je na operativnom sustavu Arch Linux, korištenjem Mono platforme kao implementacije .NET *frameworka*. Rješenje je testirano na Microsoft Windows. Modificirano korisničko sučelje programa prikazano je na slici 4.1.



Slika 4.1. Modificirano korisničko sučelje programa

Implementacija se sastojala od dva glavna dijela: implementacija/proširenje statičke

funkcije evaluacije te implementacija algoritama pretrage. Osim navedenog, učinjena su i razna druga poboljšanja, uključujući:

- dodavanje detekcije mat i pat pozicija
- mogućnost odabira algoritma te dubine pretrage
- postojeća mjera vrednovanja pozicije promijenjena je iz *floating–point* broja u cjelobrojnu korištenjem *centipawns* mjere
- prilikom odabira poteza iz popisa poteza, označena je figura koja izvodi potez na ploči kao i polje gdje će se potez izvesti
- dvoklik na potez u popisu poteza izvršava odabrani potez
- ispis poteza pretvoren je u RAN (*Reversible Algebraic Notation*) notaciju u grafičkom sučelju radi lakše vizualne provjere rezultata

Što se tiče pretrage, u programu je moguće birati između tri algoritma:

1. *negamax*
2. α–β
3. α–β, s početkom rezanja od dubine 1 nadalje,

kao što je vidljivo na slici 4.1. Algoritam 2 i 3 nude se kao posebne mogućnosti zbog tog što korištenjem α–β ne možemo napraviti procjenu vrijednosti za svaki potez kao što možemo za *negamax*, nego samo za najbolji, budući da zbog rezanja dolazi do gubitka informacija. Međutim, ako rezanje započnemo jednu razinu dublje, tada možemo procijeniti vrijednost svakog poteza. Na taj način mogu se usporediti rezultati 1. i 3. algoritma što nam omogućuje dodatnu provjeru točnosti oba uz gubitak performansi.

Implementirana funkcija evaluacije sastoji se od procjene materijala, procjene mobilnosti te procjene rasporeda (s posebnim slučajem procjene strukture pješaka). Vrijednost figura koje procjena materijala pripisuje pojedinoj figuri prikazana je u tablici 4.1 i utemeljena je na funkciji evaluacije koju je razvio Tomasz Michniweski[17].

Vrijednost kralja odnosi se na vrijednost mat pozicije, odnosno, ukoliko algoritam pronađe mat, dodjeljuje mu vrijednost kralja. Toj vrijednosti se zatim oduzima dubina (u polupotezima) na kojoj je mat pronađen kako bi algoritam preferirao završetak igre u manje poteza. Dodatno, u sklopu provjere materijala implementiran je bonus za par lovaca koji iznosi 50.

Procjena strukture pješaka uzima u obzir uduplane, izolirane i zaostale pješake. Raspored figura vrednuje se na temelju *piece–square* tablica opisanih u [17]. Iznimno, uzima se u obzir i je li kralj u šahu, što vrijedi 125 *centipawns* i služi kao jednostavna procjena za sigurnost

Tip figure	Vrijednost figure
Pješak	100
Skakač	320
Lovac	330
Top	500
Kraljica	900
Kralj	20000

Tablica 4.1. Materijalna vrijednost figura

kralja, te nalazi li se top u poluotvorenom ili otvorenom stupcu, što vrijedi 8 i 16 *centipawns*. Implementirana je i procjena faze igre koja se koristi za interpolaciju između *piece–square* tablica za srednju i završnu igru kralja. Procjena faze je općenita te bi se mogla iskoristiti za implementaciju i drugih ponašanja koja ovise o fazi igre.

4.1 Rezultati

Implementacijom algoritma s α–β rezanjem omogućena je pretraga do dubine 6–7 polupoteza u trajanju do 10 sekundi u većini pozicija, što je zadovoljavajuće s obzirom da postoji puno prostora za optimizaciju. Rezultati su dobiveni na računalu s Intel Core i7–3520M procesorom.

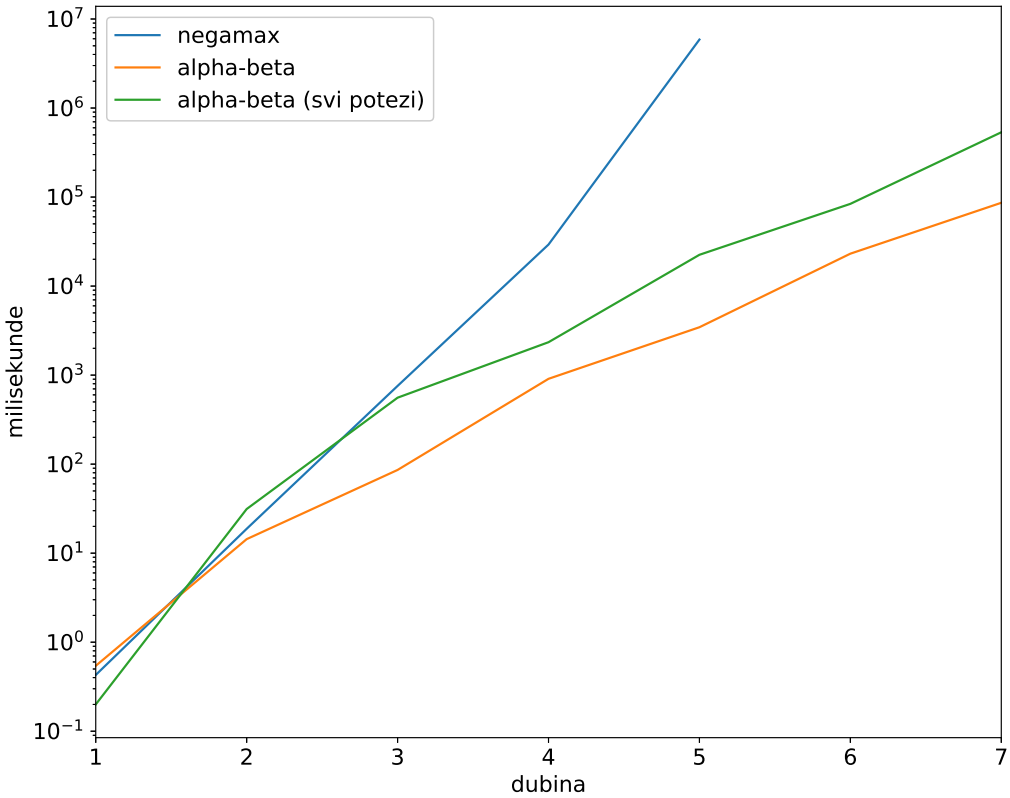
4.1.1 Performanse

Performanse programa su testirane tako da su sva tri algoritma za pretragu (*negamax*, α–β, α–β (svi potezi)) pokrenuta nad 7 različitih pozicija. Pozicije su odabrane tako da pokrivaju ranu, srednju i kasnu igru, uključujući i početnu šahovsku poziciju.

Mjerene pozicije (FEN):

1. rnbqkbnr/pppppppp/8/8/8/8/PPPPPPPP/RNBQKBNR w KQkq -
2. 8/5r2/B1N2p1k/RP1Pp3/4P2P/B4P2/3p2b1/1K6 w - -
3. 3q4/1r2Q1P1/3P1rn1/ppK4P/P1R5/5k2/N1B5/8 w - -
4. 1Q6/8/k5p1/pp6/8/3r4/4n1K1/8 w - -
5. 8/8/2p5/1r6/8/2K5/4b3/6qk w - -
6. r2qr1k1/bp1b1pp1/p1pp1nnp/P3p3/3PP3/2P2N1P/1PBN1PP1/R1BQR1K1 w - -
7. 5rk1/2pb2p1/p2q3p/P2nrp2/2Qb4/1R5P/1P1B1PP1/1B2RNK1 b - -

Svaki algoritam pokrenut je na svakoj poziciji 5 puta na različitim dubinama te je vrijeme izvršavanja izmjereno u milisekundama. Na većim dubinama (5–7), mjerenje za *negamax* algoritam bilo je nepraktično sporo pa nije uključeno u rezultate. Nad izmjerenim vremenima za svaku dubinu izračunata je aritmetička sredina te je iz rezultata dobiven graf na slici 4.2. Za y os korišteno je logaritamsko skaliranje kako bi eksponencijalan odnos bio bolje vidljiv.



Slika 4.2. Prosječna vremena pretrage za različite dubine nad 7 testnih pozicija

Sva tri algoritma pokazuju eksponencijalni porast, kako je i očekivano, no kad je uključeno α–β rezanje, vidljiv je značajno manji nagib pravca. Također možemo primjetiti da su pravci za dvije varijante α–β algoritma samo vertikalno translantirane verzije jedan drugog, i to za otprilike jedan polupotez. To je također očekivano budući da je jedina razlika među njima što, u slučaju kad vrednujemo sve poteze, započinjemo s rezanjem jedan polupotez dublje, čime gubimo prilike za rezanjem (pa tako i prilike za uštedu) na dubini 1.

4.1.2 Testiranje pretrage

Zanimljiv način testiranja algoritama pretrage je rješavanje zagonetki prisiljenog matiranja u N poteza. Budući da u toj situaciji znamo da rješenje postoji, pretraga na dubini

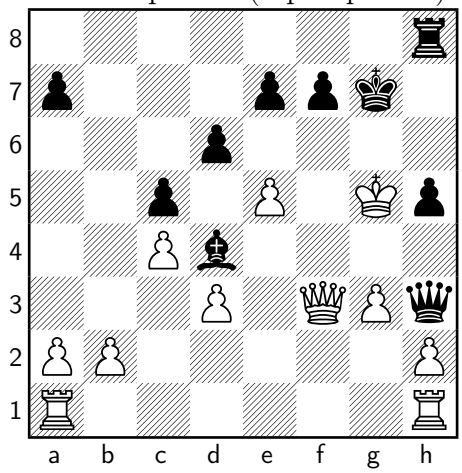
$D \geqslant N$ morala bi pronaći rješenje, ukoliko je implementacija točna. Dodatno, možemo provjeriti da algoritmi daju isto rješenje.

Provjera je učinjena nad nekoliko takvih zagonetki preuzetih sa stranice Spark Chess[18]. Ponekad je testiranje *negamax* algoritmom na punoj potrebnoj dubini bilo nepraktično zbog spore pretrage te je u tom slučaju naznačena manja dubina na kojoj je test obavljen. Algoritam s α–β rezanjem pronašao je rješenje u svim slučajevima.

Rješenja koja je proizveo šahovski program navedena su u RAN notaciji jer tu notaciju program koristi da bi ispisao poteze.

Monterinas vs Max Euwe, Amsterdam, 1927.

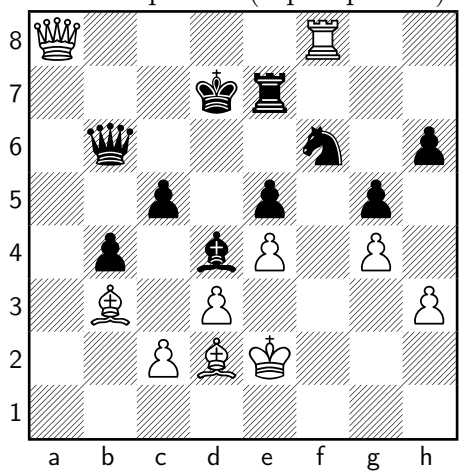
Mat u 2 poteza (3 polupoteza)



Rješenje: **1. . . ♗e3+ 2 ♕×e3 ♕g4#**
negamax: 1. . . bd4-e3 2. Qf3×be3 qh3-g4#
α–β: 1. . . bd4-e3 2. Qf3×be3 qh3-g4#

Joseph Blackburne vs Adolf Anderssen, 1873.

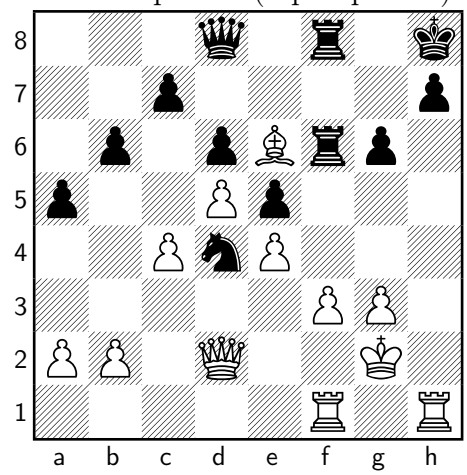
Mat u 3 poteza (5 polupoteza)



Rješenje: **1 ♕c8+ ♔d6 2 ♖×f6+ ♖e6 3 ♖×e6#**
negamax: 1. Qa8-c8 kd7-d6 2. Rf8×nf6 re7-e6 Rf6×re6#
α–β: 1. Qa8-c8 kd7-d6 2. Rf8×nf6 re7-e6 Rf6×re6#

Daniel Harrwitz vs Bernhard Horwitz, London, 1846

Mat u 3 poteza (5 polupoteza)

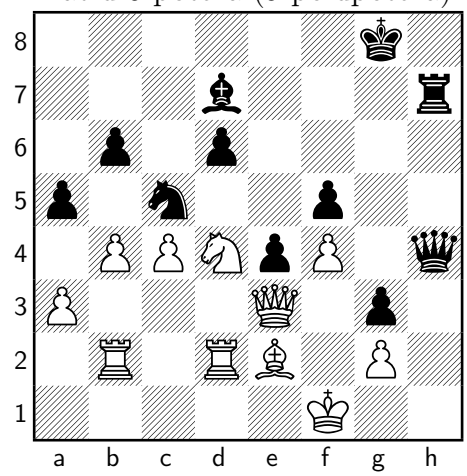


Rješenje: **1 ♖×h7 ♔×h7 2 ♖h1 ♔g7 3 ♕h6#**
negamax (dubina 4): 1. Be6-g4 a5-a4 2. b2-b3 a4×Pb3 3. Qd2-h6 rf8-f7 4. a2×pb3 nd4×Pb3 5. Qh6-e3 nb3-d4 6. Rf1-f2 nd4×Pf3 7. Rf2×nf3 rf6×Rf3 8. Bg4×rf3 qd8-a8 9. Qe3-b3 qa8-a5 10. Qb3-b2 qa5-ca 11. Rh1-a1 kh8-g7 12. Qb2-c3 qc5-d4 ...
α–β: 1. Rh1×ph7 kh8×Rh7 2. Rf1-h1 kh7-g7 3. Qd2-h6#

Kao što možemo primjetiti, *negamax* ovdje ne uspijeva pronaći rješenje zbog ograničene dubine i to za samo 1 polupotez premalo. Štoviše, ukoliko pustimo program da odigra do kraja pri dubini pretrage 4, crni matira bijelog s dvije kraljice. To nam jasno pokazuje značaj dubine pri pretrazi.

Paul Keres vs. Tigran V. Petrosian, Bled 1959

Mat u 5 poteza (9 polupoteza)



Rješenje: **1. . . ♕×f4 2 ♗f3 ♕×e3 3 ♗h5 ♖×h5 4 ♔f3 ♖h1 5 ♔g1 ♖g1**#
negamax (dubina 4): 1. . . qh4×Pf4 2. Be2-f3 qf4×Qe3 3. Bf3-h5 rh7×Bh5 4. Nd4-f3 rh5-h1 5. Nf3-g1 rh1×Ng1#
α–β: 1. . . qh4×Pf4 2. Be2-f3 qf4×Qe3 3. Bf3-h5 rh7×Bh5 4. Nd4-f3 rh5-h1 5. Nf3-g1 rh1×Ng1#

4.2 Nedostaci i moguća proširenja

S obzirom da iz naoko jednostavnih pravila šaha nastaje vrlo kompleksna igra koju nije moguće ezgaktno riješiti, šahovski programi također mogu biti vrlo kompleksni te smo s ovom implementacijom tek zagrebali površinu onog što se može postići.

Trenutni program postiže dubinu pretrage do 6–7 polupoteza u razumnom vremenu (ispod 10 sekundi), što je zadovoljavajuće s neoptimiziranom implementacijom, ali je svakako rezultat na kojem bi se moglo poraditi. Jedna očita taktika za postizanje boljih rezultata na ovom polju (osim općenite optimizacije programa) bila bi implementacija ranije opisanih transpozicijskih tablica. To bi otvorilo put prema implementaciji *quiescence* pretrage, što je vrlo bitno za šahovski program kako bi se riješio problem horizonta. Zatim bi se mogle implementirati neke od heuristika opisanih u poglavlju o funkciji pretrage, što bi dodatno povećalo stratešku jačinu programa. Strateško znanje programa moglo bi se povećati dodatnom razradom funkcije evaluacije, npr. uključivanjem boljih procjena sigurnosti kralja ili pronalaženjem boljih vrijednosti za parametre *position–square* tablica. Jedna od opcija za pronalaženje boljih parametara mogla bi uključivati mašinsko učenje nad zapisima odigranih šahovskih partija koje su dostupne u velikom broju. Konačno, performanse bi se mogle dodatno povećati paralelizacijom postojećih algoritama pretrage ili implementacijom nekog od naprednijih algoritama, poput Mtd(f).

5. ZAKLJUČAK

Cilj ovog rada je opisati princip rada te konstruirati šahovski program koji na temelju pretrage stabla igre (odnosno mogućih poteza) i vrednovanja pojedinačnih šahovskih pozicija preporučuje najbolji potez u danoj šahovskoj poziciji. Ovakav postupak je motiviran nemogućnošću rješavanja cjelokupnog stabla igre zbog kombinatorne eksplozije šaha. U radu su opisani principi rada funkcije pretrage i raznih algoritama koji objavljaju tu zadaću, zatim statičke funkcije evaluacije za pojedinačne pozicije te konačno princip rada i prednosti *bit-board* metode u ovakvom programu. Za svaki od ovih dijelova navedene su i opisane tehnike koje se mogu koristiti pri implementaciji pojedine komponente kako bi program bio strateški jači ili brži. Kao praktični dio, implementiran je šahovski program zasnovan na podskupu ovih tehnika, uključujući *negamax* algoritam s α–β rezanjem te *piece-square* tablicama. Jedan od fokusa pri implementaciji bio je naknadna proširivost programa. Programsko rješenje je testirano za točnost i performanse, a zatim su navedeni neki nedostaci i moguća proširenja.

Literatura

[1] M. Amidžić, *Šahovski računalni program*, mag. rad, Elektrotehnički fakultet Osijek, 2015.

[2] *Rohada*, http://www.sahklube4.hr/rohada/ [kolovoz 2017].

[3] *En passant*, http://www.sahklube4.hr/en_passant/ [kolovoz 2017].

[4] K. Binmore, *Playing for Real*, Oxford University Press Inc (Verlag), 2007, ISBN: 978-0-19-530057-4.

[5] George Heineman, Gary Pollice i Stanley Selkow, *Algorithms in a Nutshell, drugo izdanje*, O'Reilly Media, Inc., 2008, ISBN: 9780596516246.

[6] C. Shannon, *Programming a Computer for Playing Chess*, *Philosophical Magazine*, 7. serija 41.314 (1950).

[7] M. Strejczek, *Some aspects of chess programming*, disertacija, Technical University of Łódź, Faculty of Electrical i Electronic Engineering, Department of Computer Science, 2004.

[8] *Computer chess programming theory*, http://www.frayn.net/beowulf/theory.html [srpanj 2017].

[9] G. C. Fox, R. D. Williams i P. C. Messina, *Parallel Computing Works*, Morgan Kaufmann Publishers, 1994, ISBN: 1-55860-253-4, http://www.netlib.org/utk/lsi/pcwLSI/text/node351.html.

[10] J. Pearl i R. E. Korf, *Search Techniques*, *Annual Review of Computer Science* 2.1 (1987), str. 451–467, DOI: 10.1146/annurev.cs.02.060187.002315, eprint: https://doi.org/10.1146/annurev.cs.02.060187.002315, https://doi.org/10.1146/annurev.cs.02.060187.002315.

[11] A. Plaat i dr., *SSS* = Alpha-Beta + TT*, (2014), arXiv: arXiv:1404.1517, https://arxiv.org/abs/1404.1517.

[12] A. Plaat i dr., *A New Paradigm for Minimax Search*, teh. izv., 1994.

[13] D. Beal, *A Generalised Quiescence Search Algorithm*, *Artif. Intell.* 43.1 (travanj 1990), str. 85–98, ISSN: 0004-3702, DOI: 10.1016/0004-3702(90)90072-8, http://dx.doi.org/10.1016/0004-3702(90)90072-8.

[14] *Aspiration window*, https://chessprogramming.wikispaces.com/Aspiration+Windows [lipanj 2017].

[15] *Null Move Pruning*, https://chessprogramming.wikispaces.com/Null+Move+Pruning [lipanj 2017].

[16] J. Condon i K. Thompson, *BELLE*, (1983).

[17] Tomasz Michniewski, *Simplified evaluation function*, https://chessprogramming.wikispaces.com/Simplified+evaluation+function [lipanj 2017].

[18] *Spark Chess, chess puzzles*, https://www.sparkchess.com/chess-puzzles.html [kolovoz 2017].

SAŽETAK

U ovom radu opisani su ključni dijelovi i principi rada šahovskih programa koji pretražuju i vrednuju moguće poteze te na temelju toga procjenjuju najbolji potez. Fokus rada je na šahovske programe bazirane na *bitboard* metodi zapisa pozicije. Promotreni su različiti algoritmi pretrage poteza te pripadajuće heuristike koje se koriste u modernim šahovskim programima. Opisana je problematika vrednovanja šahovskih pozicija te tehnike koje se koriste pri istoj. U sklopu rada implementiran je šahovski program koji pomoću *negamax* algoritma s α–β rezanjem i drugih opisanih tehnika analizira poziciju te procjenjuje najbolji potez. Kao programska podloga korišteno je prethodno razvijeno rješenje koje nudi generator legalnih poteza te grafičko sučelje. Na kraju rada programsko rješenje je testirano te su mu izmjerene performanse. Navedeni neki nedostaci i moguća buduća proširenja.

Ključne riječi: šah i računala, minimax, negamax, α–β rezanje, *bitboard*, vrednovanje poteza, pretraga poteza, teorija igara

ABSTRACT

Evaluation of available chess moves using the bitboard method

The thesis describes key parts and working principles of chess engines, which search and evaluate possible moves in order to give an estimate of the best move. The focus of the thesis were chess engines based on the bitboard method of position encoding. Several different search algorithms and their accompanying heuristics, as used in modern chess engines, were examined. Some common tehniques and considerations when evaluating chess positions were described. A chess engine which estimates the best move using the negamax algorithm with α–β pruning, and other described techniques, was implemented as part of the thesis. A previously developed solution was used for its GUI and legal move generator as a foundation. Finally, the resultant chess engine was tested and benchmarked. Some shortcomings of the final engine were listed, along with possible improvements in future works.

Keywords: computer chess, minimax, negamax, α–β pruning, bitboard, move evaluation, move search, game theory

ŽIVOTOPIS

Denis Kasak rođen je u Osijeku, 23. studenog 1988. godine. Osnovno obrazovanje stekao je u osnovnoj školi Vijenac u Osijeku. Nakon toga, pohađao je Prirodoslovno-matematičku gimnaziju u Osijeku, a zatim upisao studij računarstva na Elektrotehničkom fakultetu Osijek.

Programiranjem se bavi od trećeg razreda osnovne škole kad se susreo s programskim jezikom Logo u osječkom Domu Tehnike, a danas mu je najdraži programski jezik Haskell. Dvaput je sudjelovao u programu Google Summer of Code: prvi put u sklopu organizacije Python Software Foundation na implementaciji *sandboxa* za *tinypy*, minimalističkoj implementaciji jezika Python, a drugi put u sklopu organizacije ScummVM, na reimplementaciji stare češke *point-and-click* avanture *Dračí Historie*. Trenutno se bavi *freelancingom* i *bug bountyjima*.

Interesi mu uključuju funkcionalno programiranje, formalnu verifikaciju, sustave tipova (*type systems*), primjenu apstraktne algebre i teorije kategorija u programiranju, računalnu sigurnost, *fuzzing* i *reverse engineering*.

PRILOZI

Na CD-u:

- Diplomski rad u PDF formatu
- Izvorni kod programskog rješenja