UNIVERZA V MARIBORU

Fakulteta za elektrotehniko, računalništvo in informatiko

ANALIZA ALGORITMA ZA IGRANJE ŠAHA

Seminarsko opravljan izpit:
NAČRTOVANJE IN ANALIZA ALGORITMOV

Maribor, junij 2005

Študent: Borko Bošković

Kazalo

1. Predstavitev algoritma

Iskalni algoritem je osrednji del šahovskih programov. Le ta na določen način preiskuje iskalno drevo s pomočjo ocenitvene funkcije, na koncu pa poda potezo za nadaljevanje igre. Z matematičnega stališča lahko igro šah opišemo kot igro med dvema nasprotnikoma s popolno informacijo in ničelno vsoto [1]. Kot vemo je šah igra med dvema igralcema in sicer med belim in črnim. Oba igralca imata popoln pregled nad celotno igro (pozicijo in potezami), zato takim igram pravimo tudi igre s popolno informacijo. Dodatno lastnost šahovske igre pa predstavlja "ničelna vsota". Določa jo strukturno ravnotežje v tekmovalni naravi igre. Igra poteka z izmenjavo potez med igralcema. Vsaka od legalnih potez prinaša določeno prednost oz. slabost za igralca na potezi oz. njegovega nasprotnika. Tako lahko npr. potezo p ovrednotimo za oba igralca (belega - $eval_w(p)$, črnega - $eval_b(p)$) in dobimo naslednji izraz oz. "ničelno vsoto":

$$eval_w(p) + eval_b(p) = 0.$$

Na osnovi tega matematičnega opisa lahko zgradimo drevo igre. To je drevo, ki v korenu drevesa vsebuje začetno pozicijo, v listih drevesa pa se nahajajo končne pozicije. Problem tega drevesa pri šahovski igri je njegova ogromna velikost. Drevo igre vsebuje približno w^d vozlišč, kjer w predstavlja povprečno število potez na eno pozicijo, d pa povprečno število potez na partijo. Tako iskalno drevo praktično ni mogoče preiskati. Zato šahovski programi uporabljajo algoritme, ki preiskujejo samo del drevesa igre. Ta del drevesa imenujemo tudi iskalno drevo. To drevo v korenu vsebuje trenutno pozicijo igre, v listih pa vozlišča, ki zadoščajo določenemu pogoju. Ta pogoj je lahko npr. določen z globino iskanja.

1.1 Algoritem MINIMAX

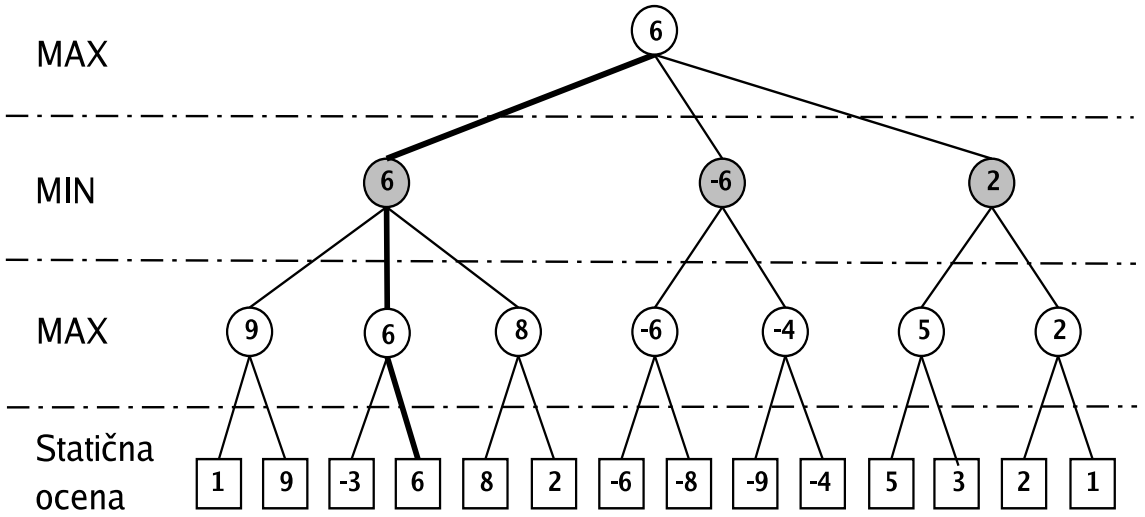
Igralca v šahovski igri izmenjujeta poteze in oba poskušata izbrati najboljšo svojo potezo oz. maksimirati svojo vrednost, posledično pa minimizirati nasprotnikovo. Tako glede na trenutno pozicijo identificiramo igralca MIN in MAX. V začetni poziciji igralca MAX predstavlja beli igralec, ki je na potezi. Igralca MIN pa predstavlja črni igralec. Na osnovi opisanega lahko formuliramo algoritem za igranje iger med dvema nasprotnikoma s popolno informacijo in ničelno vsoto. To naredimo tako, da simuliramo obnašanje igralcev MIN in MAX oz. ti. načela MINIMAX.

Opisan algoritem prikazuje algoritem 1 in njegovo iskalno drevo prikazuje slika 1.1. Algoritem temelji na prej opisanem načelu MINIMAX, rekurziji in ocenitveni funkciji. Ideja algoritma je v tem, da drevo igre preiščemo samo do določene globine [1, 2, 5, 3, 7].

```
int MINIMAX(Position p, int depth){
    int best, value;
    // Pogoj za končanje rekurzije
    if( p.endGame() || depth <= 0 ) return p.evaluate();

    // Ustvarimo seznam vseh možnih potez
    Moves moves = p.generateLegalMoves();

    // Poiščemo najboljšo potezo
    if( p.sideToMove() == WHITE ){
        best = - Evaluate.INFINITY;
        // Poiščemo oceno najboljše poteze belega - MAX igralca
        while( p.makeNextMove(moves) ){
            value = MINIMAX(p,depth-1);      // Ocenimo pozicijo
            p.unmakeMove();
            if( value > best )best = value;      // Maksimiramo vrednost
        }
    }else{
        best = Evaluate.INFINITY;
        // Poiščemo oceno najboljše poteze črnega - MIN igralca
        while( p.makeNextMove(moves) ){
            value = MINIMAX(p,depth-1);      // Ocenimo pozicijo
            p.unmakeMove();
            if( value < best )best = value;      // Minimiziramo vrednost
        }
    }
    return best;
}
```

Algoritem 1: MINIMAX

Liste (terminalna vozlišča) tako zgrajenega drevesa ocenimo z ocenitveno funkcijo. Ocenitvena funkcija v kontekstu MINIMAX vrne zelo veliko pozitivno vrednost, če je črni matiran, zelo veliko negativno vrednost, če je beli matiran in nič, če je pozicija remi. Če pozicija ne predstavlja konca igre, se vrne hevristična ocenitev. Hevristična ocenitev bo vedno pozitivna, če beli zmaguje oz. negativna, če črni zmaguje in v okolici ničle, če je pozicija enakopravna za oba igralca. Dobljene ocene končnih pozicij, s pomočjo ocenitvene funkcije, "potujejo" navzgor po drevesu v skladu z načelom MINIMAX. Tako se ovrednotijo vse pozicije. Potezo, ki pa jo v izhodiščni poziciji (korena drevesa) izberemo po načelu MINIMAX, predstavlja izbrano potezo iskalnega algoritma. Iskalno drevo opisanega algoritma in ocenitev vozlišč prikazuje slika 1.1.



Slika 1.1: Iskalno drevo algoritma MINIMAX

1.2 Algoritem NEGAMAX

Algoritem MINIMAX lahko optimiziramo tako, da spremenimo predznak vrednosti nasprotnikove ocenitve. Tako se rezultat ocenitve nasprotnika preslika v oceno trenutnega igralca. Ta optimizacija zahteva tudi spremembo v ocenitveni funkciji. Ocenitvena funkcija v tem kontekstu vrne pozitivno vrednost, če je pozicija boljša za igralca na potezi oz. negativno, če je pozicija boljša za igralca, ki ni na potezi. Opisan algoritem se imenuje NEGAMAX in prikazuje ga algoritem 2. V primerjavi z MINIMAX algoritmom pa vsebuje dosti manj kode, zmanjšuje število napak pomnilniških strani in omogoča lažje vzdrževanje (dodajanje in odstranjevanje funkcionalnosti) programa [1, 5, 3].

Prikazana algoritma sta relativno enostavna in neučinkovita. Temeljita na MINIMAX načelu in sistematično preiskujeta celotno iskalno drevo. Ker algoritma preiskujeta celotno iskalno drevo, je časovna zahtevnost w^d. Vejitveni faktor v iskalnem drevesu oz. povprečno število nadaljevanj v poziciji je 35. Tako je npr. za globino 6 potrebno preiskati drevo velikosti $1.8 \cdot 10^9$ vozlišč ali za globino 10 drevo $2.8 \cdot 10^{15}$ vozlišč.

1.3 Algoritem alfa–beta

Alfa–beta algoritem predstavlja izboljšavo prejšnjih dveh algoritmov. Izboljšava temelji na tem, da za pozicijo, o kateri vemo, da je slabša od trenutno najboljše izbrane, ne ugotavljamo za koliko je slabša. To idejo formaliziramo tako, da vpeljemo dve meji alfa in beta. Alfa predstavlja najslabši rezultat, ki je za igralca na potezi že zagotovljen in vse kar je manjše ali enako od te meje, ne omogoča izboljšave. Beta pa predstavlja

```
int NEGAMAX(Position p, int depth){
    int best = - Evaluate.INFINITY, value;
    // Pogoj za končanje rekurzije
    if( p.endGame() || depth <= 0 ) return p.evaluate();

    // Ustvarimo seznam vseh možnih potez
    Moves moves = p.generateLegalMoves();

    // Poiščemo najboljšo potezo
    while( p.makeNextMove(moves) ){
        value = -NEGAMAX(p,depth-1);      // Ocenimo pozicijo
        p.unmakeMove();
        if( value > best )best = value;          // Maksimiramo vrednost
    }
    return best;
}
```

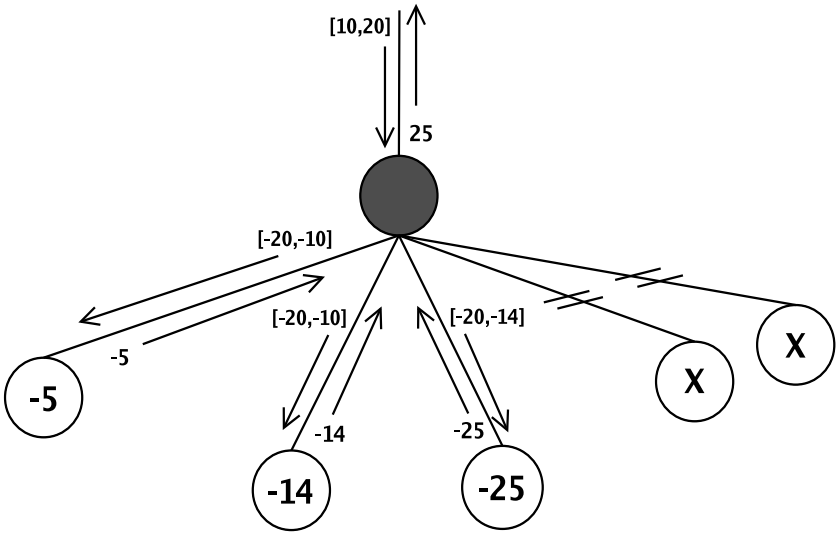
Algoritem 2: NEGAMAX

najslabši scenarij za nasprotnika, ki mu je že zagotovljen [8]. Tako v primeru, ko ima trenutna pozicija večjo ali enako vrednost kot beta, sigurno ne bo del glavne variante (Principal Variation) oz. del seznama izbranih potez od korena do lista drevesa. Zato te pozicije ni potrebno več preiskovati.

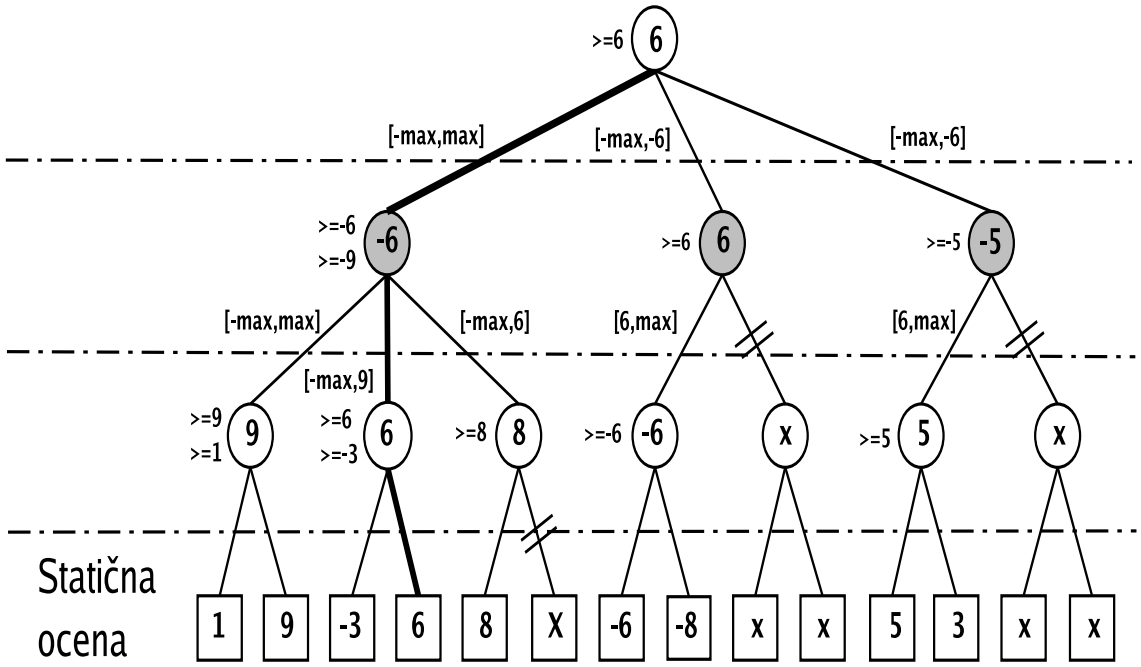
Poglejmo opisan algoritem na primeru vozlišča, ki ga prikazuje slika 1.2. Vozlišče prejme meji alfa in beta oz. iskalno okno (search window) 10,20. Ker algoritem temelji na NEGAMAX algoritmu se preiskovanje nadaljuje z oknom -20,-10. Ocenitev prvega vozlišča je -5. Rezultatu spremenimo predznak in dobimo manjšo vrednost od spodnje meje iskalnega okna. Tako nadaljujemo iskanje v drugem vozlišču. To vozlišče vrne vrednost -14. Ponovno spremenimo predznak in dobimo vrednost znotraj iskalnega okna. Iskalno okno zdaj spremenimo na 14,20. Preiskovanje nadaljujemo v tretjem vozlišču. To vozlišče vrne vrednost -25. S spremembo predznaka dobimo vrednost, ki je večja od zgornje meje iskalnega okna. To pa pomeni, da je ocena vozlišča slabša od trenutno najboljšega izbranega vozlišča v starševskem vozlišču. Ker nas ne zanima, za koliko je ocena tega vozlišča slabša, lahko končamo z ocenitvijo tega vozlišča. Temu pravimo rez (cutoff) in njegova posledica je manjše iskalno drevo.

Zmogljivost algoritma alfa-beta je zelo odvisna od zaporedja preiskanih potez. Če najprej preiščemo poteze, ki povzročijo reze, dobimo znatno manjše iskalno drevo. V najslabšem primeru algoritem deluje enako kot MINIMAX algoritem in preiskuje celotno iskano drevo. V primeru najboljšega zaporedja, algoritem znatno zmanjša iskalno drevo oz. oblikuje minimalno drevo (minimal tree). Velikost minimalnega drevesa pa je:

$$w^{\lceil \frac{d}{2} \rceil} + w^{\lfloor \frac{d}{2} \rfloor} - 1$$



Slika 1.2: Vozlišče alfa–beta algoritma



Slika 1.3: Iskalno drevo algoritma alfa-beta

Z računalniškega stališča alfa-beta algoritem v najboljšem primeru reducira časovno zahtevnost z $O(w^d)$ na $O(\sqrt{w^d})$. Tako alfa–beta algoritem v povprečju zmanjša vejitveni faktor s 35 na približno 6 [1] in omogoča, da v enakem času preiščemo drevo z večjo globino. Alfa–beta algoritem prikazuje algoritem 3. Primer iskalnega drevesa in način delovanja algoritma pa prikazuje slika 1.3.

```
int alphaBeta(Position p, int depth, int alpha, int beta){
    int value;
    // Pogoj za končanje rekurzije
    if( p.endGame() || depth <= 0 ) return p.evaluate();

    // Ustvarimo seznam vseh možnih potez
    Moves moves = p.generateLegalMoves();

    // Poiščemo najboljšo potezo
    while( p.makeNextMove(moves) ){
        value = -alphaBeta(p,depth-1,-beta,-alpha);      // Ocenimo pozicijo
        p.unmakeMove();

        // Preverimo ali vozlišče pripada glavni varianti
        if( value > alpha ){
            if( value >= beta ) return value;
            alpha = value;
        }
    }
    return alpha;
}
```

Algoritem 3: Alfa–beta

1.4 Iterativno poglabljanje

Pri šahu je eden od zelo pomembnih dejavnikov čas. Tako se pojavi vprašanje, kako v določenem času poiskati najboljšo potezo? Možna rešitev je, da najprej preiskujemo z globino 1, če smo končali pred iztekom časa, začnemo preiskovati z globino 2, če tudi po tem iskanju ni potekel čas, nadaljujemo iskanje z globino 3 itd., dokler ne poteče čas. Opisani algoritem se imenuje iterativno poglabljanje (Iterative Deepening) in prikazuje ga algoritem 4.

```
int iterativeDeepening(Position p){
    int value;
    for( int depth=1;;depth++ ){
        value = alphaBeta(p,depth,-Evaluate.INFINITY, Evaluate.INFINITY);
        if( timeOut() ) break ;// Če je čas potekel,iskanje končamo
    }
    return value;
}
```

Algoritem 4: Iterativno poglabljanje

Na prvi pogled se zdi, da je to iskanje zelo počasno. Če pa uporabljamo transpozicijsko

tabelo (podpoglavje 1.8), se ta algoritem izkaže kot hitrejši od navadnega alfa-beta algoritma. Razlog temu je zaporedje dobro urejenih potez. Pri manjših globinah iskanja se transpozicijska tabela napolni z dobrimi potezami. Vsebino transpozicijske tabele pa nato uporabimo pri večjih globinah. Tako dobimo relativno dobro zaporedje potez, ki pa omogoča alfa-beta algoritmu znatno hitrejše preiskovanje [3].

1.5 Aspiracijsko iskanje

Iterativno poglabljanje uporablja alfa-beta algoritem z zelo velikim oknom (+/− INFINITY). To okno lahko zmanjšamo in spreminjamo glede na preiskovanja v prejšnjih iteracijah. Kajti verjetnost, da bo rezultat preiskovanja v prejšnji iteraciji podoben rezultatu pri naslednji iteraciji, je relativno visoka. Temu algoritmu pravimo aspiracijsko iskanje (Aspiration Search) in prikazuje ga algoritem 5.

```
int aspirationSearch(Position p){
    int margin = 50; // Določa velikost iskalnega okna
    int alpha = -Evaluate.INFINITY;
    int beta = Evaluate.INFINITY;

    for( int depth=1;;depth++ ){
        value = alphaBeta(p,depth,alpha, beta);

        // Če je čas potekel, iskanje končamo
        if( timeOut() ) break ;

        // Če je rezultat manjši od spodnje meje moramo iskanje ponoviti
        if( value <= alpha )
            value = alphaBeta(p,depth,-Evaluate.INFINITY,value);

        // Če je rezultat večji od zgornje meje moramo iskanje ponoviti
        else if( value >= beta )
            value = alphaBeta(p,depth,value,Evaluate.INFINITY);

        // Določimo novo iskalno okno
        alpha = value - margin;
        beta = value + margin;
    }
    return value;
}
```

Algoritem 5: Aspiracijsko iskanje

Prikazani algoritem je zasnovan tako, da najprej začne preiskovati z globino 1 in maksimalno velikostjo okna. Pridobljeni rezultat iskanja nato uporabi za oblikovanje velikosti

novega iskalnega okna. S tem oknom pa nadaljujemo iskanje v naslednji iteraciji. V primeru ko se rezultat tega iskanja nahaja izven iskalnega okna, je iskanje potrebno ponoviti z ustreznim spremenjenim oknom. V nasprotnem primeru pa ponovno glede na oceno, določimo novo okno in nadaljujemo iskanje v naslednji iteraciji.

Pri tem iskanju se pojavi vprašanje, kako široko okno potrebujemo za optimalno iskanje. Na eni strani premala okna povzročajo ponovna iskanja in upočasnijo algoritem. Na drugi strani pa prevelika okna pravtako povzročajo počasnejše iskanje.

1.6 Iskanje mirovanja

Alfa-beta algoritma temelji na preiskovanju do določene globine. V primeru, ko parameter globine dobi vrednost nič, se iskanje konča in pozicija se oceni s pomočjo ocenitvene funkcije. Ocenitvena funkcija je hevristična funkcija, ki zaradi dinamične in kompleksne šahovske igre ne zna oceniti dinamičnih pozicij oz. pozicij, ki vsebujejo jemanja in promocije. Zaradi tega lahko alfa-beta algoritem izbere zelo slabo potezo. Tej lastnosti algoritma pravimo tudi učinek obzorja (Horizon Effect) ali "kratkovidnost". Izognemo se ji tako, da uporabimo algoritem iskanja mirovanja (Quiescent Search).

Iskanje mirovanja v alfa-beta algoritmu uporabimo tako, da nadomestimo klic ocenitvene funkcije s klicem iskanja mirovanja. Ta algoritem pri ocenjevanju dinamičnih pozicij nadaljuje preiskovanje do statičnih pozicij. Te pozicije pa oceni s pomočjo ocenitvene funkcije. Primer iskanja mirovanja prikazuje algoritem 6.

Ta algoritem je na prvi pogled zelo podoben alfa-beta algoritmu, toda razlike so zelo pomembne. Algoritem, preden nadaljuje iskanje v sinovih, izračuna statično oceno in po principu alfa-beta algoritma ugotovi, ali pozicija predstavlja rez. Tako se izognemo "napihnjenim iskalnim drevesom" oz. eksploziji iskanja mirovanja (quiescent search explosion). Če je ocenjena pozicija še vedno lahko del glavne variante, se preiskovanje nadaljuje podobno kot pri alfa-beta algoritmu. Razlika je le v tem, da ne ustvarimo seznama vseh potez, ampak seznam vseh potez jemanj in promocij. Za odkrivanje mata pa algoritmu moramo dodati še preverjanje, ali se trenutna pozicija nahaja v šahu. Če se pozicija nahaja v šahu, iskanje nadaljujemo s pomočjo alfa-beta algoritma in globino iskanja 1.

Tukaj je potrebno poudariti, da algoritmi iskanja mirovanja morajo biti hitri oz. ne smejo povzročati eksplozije iskanja. Da bi se temu izognili, algoritmi na različne načine klestijo iskalno drevo. Posledica tega pa je nepravilna ocena za določene pozicije, kar lahko povzroči nestabilna iskanja.

```
int quiescentSearch(Position p, int alpha, int beta){
    // V primeru šaha preiskovanje nadaljujemo z globino 1 in
    // alfa-beta algoritmom
    if( p.inCheck() ) return alphaBeta(p,1,alpha,beta);

    // Ocenimo pozicijo in ukrepamo po principu alfa-beta algoritma
    int value = p.evaluate();
    if( value > alpha ){
        if( value >= beta ) return value;
        alpha = value;
    }
    // Ustvarimo seznam vseh možnih jemanj in promocij
    Moves moves = p.generateAllCaptures();

    // Poiščemo najboljšo potezo
    while( p.makeNextMove(moves) ){
        value = - quiescentSearch(p,-beta,-alpha);
        p.unmakeMove();
        if( value > alpha ){
            if( value >= beta ) return value;
            alpha = value;
        }
    }
    return alpha;
}
```

Algoritem 6: Iskanje mirovanja

1.7 Klestenje z ničelno potezo

Za povečanje globine iskanja, s tem pa tudi zmogljivosti programov, lahko uporabimo selektivne algoritme. Ti algoritmi na račun doseganja večjih globin dodatno klestijo iskalno drevo z možnostjo, da določena pomembna vozlišča spregledajo. Primer takega algoritma je klestenje z ničelno potezo (Null-Move Pruning). Ideja algoritma je, da nasprotniku dovoli odigrati potezo namesto igralca na potezi (ničelna poteza). Nato pa preveri, ali je pozicija za igralca na potezi še vedno dovolj dobra. Tako v primeru, da je iskanje s pomočjo ničelne poteze za igralca na potezi še vedno dovolj dobro, lahko predčasno končamo s preiskovanjem in znatno pohitrimo algoritem.

Pri tem algoritmu se moramo zavedati, da v določenih primerih klestenje odpove. Določene pozicije se lahko napačno ocenijo in povzročijo izbiro slabe poteze. Primer takšnih pozicij so pozicije v šahu. Pri teh pozicijah, kadar odigramo ničelno potezo, dobimo neveljavno pozicijo. Zato je iskanje mirovanja potrebno omejiti le na pozicije, ki niso v šahu. Dodaten problem predstavljajo še primeri, kadar se zaporedoma odigrajo samo ničelne poteze. Tako dobimo degradirano iskanje. Rešitev tega problema

je omejevanje izvajanja dveh zaporednih ničelnih potez.

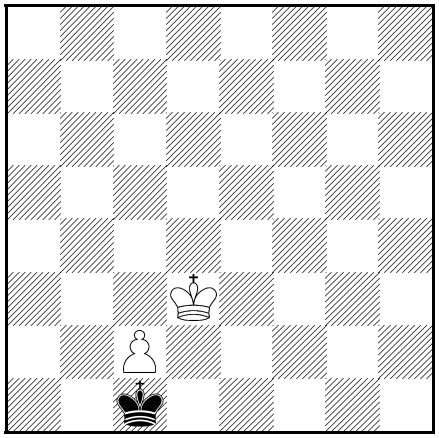
Klestenje pri tem algoritmu predstavlja reduciranje globine iskanja pri iskanju z ničelno potezo. Reduciranje temelji na zmanjšanju globine iskanja za določen faktor. V klasičnem iskanju mirovanja ima ta faktor vrednost 2, z dodatnimi izboljšavami pa lahko ima tudi vrednost 3. Tako npr., če določeno vozlišče preiskujemo z globino D in imamo definiran faktor reduciranja R, potem je globina iskanja z ničelno potezo enaka D-R.

```
int R = 2;
int alphaBeta(Position p, int depth, int alpha, int beta){
    int value;
    // Pogoj za končanje rekurzije
    if( p.endGame() || depth <= 0 ) return p.evaluate();

    // Iskanje z ničelno potezo, minimalnim oknom in faktorjem reduciranja R
    if( !p.inCheck() && depth > 1 ){
        p.makeNullMove();
        value = -alphaBeta(p,depth-1-R,-beta,-beta+1);
        p.unmakeNullMove();
        if(  value >= beta ) return beta;
    }
    // Ustvarimo seznam vseh legalnih potez
    Moves moves = p.generateLegalMoves();

    // Poiščemo najboljšo potezo enako kot v alfa-beta algoritmu
    while( p.makeNextMove(moves) ){
        value = -alphaBeta(p,depth-1,-beta,-alpha);      // Ocenimo pozicijo
        p.unmakeMove();
        if( value > alpha ){
            if( value >= beta ) return value;
            alpha = value;
        }
    }
    return alpha;
}
```

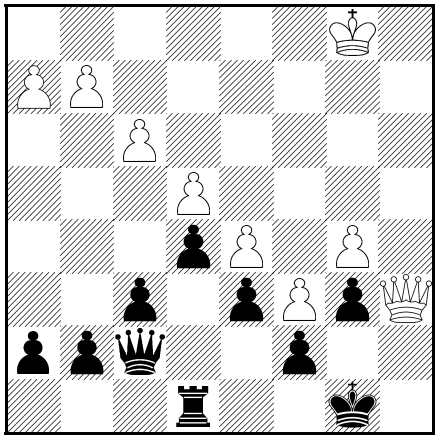
Algoritem 7: Klestenje z ničelno potezo

Algoritem iskanja z ničelno potezo prikazuje algoritem 7. Algoritem omejuje iskanje mirovanja na pozicije, ki niso v šahu. Poleg reduciranja faktorja globine nad iskanjem z ničelno potezo, uporablja še iskanje z minimalnim oknom. Dobljen rezultat iskanja pa nato preveri, ali je lahko del glavne variante. Preostali del algoritma pa je enak osnovnemu alfa-beta algoritmu.

Na žalost klestenje z ničelno potezo ne deluje v vseh primerih. Primer take pozicije je pozicija, ki jo prikazujemo v nadaljevanju. V tej potezi, če črni igralec odigra potezo Kb2, sledi poteza belega Kd2 in zmaga beli igralec. V primeru ničelne poteze, beli igralec lahko odigra Kc3 ali katero koli drugo potezo in nastala pozicija predstavlja remi pozicijo. Podoben problem nastopi tudi v primeru, če je na potezi beli igralec. Katerakoli njegova poteza pripelje v remi. S pomočjo ničelne poteze pa črni igralec mora odigrati Kb2 in tako omogoča zmago belega igralca. Take pozicije imenujemo "izsiljene" (zugzwang) in so zelo pogoste v končnicah. Zato algoritem iskanja z ničelno potezo ponavadi ne uporabljamo v končnicah.



Na spodnjem primeru si poglejmo še en problem, ki lahko nastopi v primeru klestenja z ničelno potezo. Pozicija je taka, da beli igralec v vsakem primeru matira črnega s potezo Qg2. Pri preiskovanju z globino iskanja 2, se z ničelno potezo spremeni igralec na potezi in zmanjša se globina. Ker ima globina vrednost 0, se pozicija oceni na osnovi statične ocene ter tako dobimo napačno oceno. Ta primer predstavlja horizontalni efekt iskanja z ničelno potezo. Izognemo se mu tako, da ničelne poteze omejimo na preiskovanja z večjimi globinami.



Klestenje z ničelno potezo, lahko določene pomembne pozicije v iskalnem drevesu spregleda oz. napačno oceni. Take pozicije so bolj pogoste v končnicah igre. Število pozicij,

ki jih algoritem spregleda, lahko na različne načine reduciramo. Prvi način je uporaba adaptivnega faktorja reduciranja. To pomeni, da faktor reduciranja prilagajamo tipu pozicije in globini iskanja. Drug način pa predstavlja verifikacijo klestenja z ničelno potezo.

1.8 Transpozicijska tabela

V algoritmih, ki temeljijo na alfa-beta algoritmu, je zelo pomembno zaporedje potez. S pomočjo dobrega zaporedja potez lahko znatno zmanjšamo velikost iskalnega drevesa in dosežemo večjo globino iskanja. Tako se je izkazalo, da je kvaliteta algoritma manj pomembna od kvalitete zaporedja potez. Mala sprememba pri oblikovanju zaporedja potez, lahko izboljša zmogljivost iskalnih algoritmov od 50% do 100% in več. S pomočjo uporabe statičnih in dinamičnih hevristik, lahko oblikujemo iskalno drevo, ki je le za 20% do 30% večje od minimalnega drevesa [1].

Statično hevristiko predstavlja ocenitvena funkcija, dinamično pa zbiranje informacij o že odigranih potezah. V ta namen se uporablja transpozicijska tabela.

Transpozicijska tabela je tabela, ki se uporablja za odkrivanje že preiskanih pozicij. Tako se lahko izognemo nepotrebnim ponovnim preiskovanjem. Transpozicijska tabela je zgrajena tako, da vsebuje naslednje podatke o pozicijah [6, 4]:

- ključ,
  Ključ predstavlja ključ določene pozicije in omogočajo indeksiranje transpozicijske tabele ter ločevanje dveh različnih pozicij.

- potezo,
  Je izbrana poteza v poziciji, ki jo določa ključ pozicije. Dodatno pa jo določajo še preostali parametri.

- ocenitev,
  Predstavlja ocenitev izbrane poteze. Pomen ocenitve dodatno določa tip ocenitve.

- tip ocenitve in
  Tip ocenitve dodatno opisuje izbrano potezo ter njeno ocenitev in lahko zavzame eno od naslednjih vrednosti: točen rezultat, spodnja meja in zgornja meja. Če najdemo najboljšo poteze v poziciji za določeno globino iskanja, potem jo v transpozicijsko tabelo shranimo kot točen rezultat. Če določena poteza povzroči rez pozicijo shranimo kot spodnjo mejo. V primeru, da nobena od potez ne izboljša spodnje meje iskalnega okna, pozicijo shranimo kot zgornjo mejo.

- globino.
  Globina predstavlja relativno globino preiskanega poddrevesa. Npr., kadar preiskujemo do globine n, pozicijo pa shranjujemo na globini m, tedaj je vrednost globine enaka n-m.

Transpozicijsko tabelo v iskalnih algoritmih uporabljamo tako, da najprej pogledamo, če vsebuje podatke o trenutni poziciji. Če te podatke vsebuje, se glede na vrednost globine in tipa ocenitve odločimo za naslednja nadaljevanja [4]:

- Prvi primer predstavlja stanje iskalnega algoritma, v katerem je globina iskanja manjša ali enaka od globine, pridobljene iz transpozicijske tabele in vrednost tipa ocenitve predstavlja točen rezultat. V tem primeru se iskanje ne nadaljuje. Poteza transpozicijske tabele se doda glavni varianti. Rezultat ocene preiskovane pozicije pa predstavlja ocena, pridobljena iz transpozicijske tabele.

- Drugi primer predstavlja stanje iskalnega algoritma, v katerem je globina iskanja manjša ali enaka od globine, pridobljene iz transpozicijske tabele in vrednost tipa ocenitve ne predstavlja točnega rezultata. Pridobljeno oceno pozicije nato uporabimo za določanje novega iskalnega okna. Spodnja meja iskalnega okna se določa v primeru, kadar tip ocenitve predstavlja spodnjo mejo. V primeru ko tip ocenitve predstavlja zgornjo mejo, se določa zgornja meja iskalnega okna. Nato preverimo še iskalno okno. Če ima spodnja meja večjo vrednost od zgornje meje, dobimo stanje, ki predstavlja rez in iskanja ni potrebno nadaljevati. V nasprotnem primeru, nadaljujemo preiskovanje tako, da najprej preiščemo pozicijo nastalo na osnovi poteze pridobljene iz transpozicijske tabele.

- Tretji primer predstavlja stanje iskalnega algoritma, v katerem je globina iskanja večja od globine, pridobljene iz transpozicijske tabele. V tem primeru pridobljeno potezo iz transpozicijske tabele uporabimo kot prvo potezo za nadaljnje preiskovanje. Na tak način izboljšamo kvaliteto zaporedja potez.

Z uporabo iterativnega poglabljanja in iskanja z minimalnim oknom, nam predstavljena transpozicijska tabela znatno reducira delo iskalnih algoritmov. Ta lastnost pride še posebej do izraza v šahovskih končnicah, ki na deski vsebujejo le nekaj figur. Primer uporabe transpozicijske tabele prikazuje algoritem 8.

```
int alphaBeta(Position p, int depth, int alpha, int beta){
    int oldAlpha = alpha, value, best = -Evaluate.INFINITY;
    boolean search = true ;
    Move lastMove, bestMove;
    TTMove ttMove= TTable.find(p.getKey());
    // Če pozicija ni najdena je ttMove.move = 0 in ttMove.depth = -1
    if( ttMove.Depth >= depth ){
        if( ttMove.flag == TTable.Exact ) return ttMove.eval;
        else if( ttMove.flag == TTable.Lower ) alpha = max(alpha, ttMove.eval);
        else if( ttMove.flag == TTable.Upper ) beta = min(beta, ttMove.eval);
        if( alpha >= beta ) return ttMove.eval;
    }
    // Pogoj za končanje rekurzije
    if( p.endGame() || depth <= 0 ){
        value = p.evaluate();
        TTable.record(p.getKey(),0,value,TTable.Exact,0);
        return value;
    }
    // Iskanje najprej nadaljijemo s potezo transpozicijske tabele
    if( ttMove.depth > 0 ){
        p.makeMove(ttMove.move);
        best = -alphaBeta(p,depth-1,-beta,-alpha);      // Ocenimo pozicijo
        bestMove = p.unmakeMove();
        if( best > alpha )                  // Ali poteza pripada glavnivarianti
            if( best >= beta ) search = false ;
            else alpha = best;
    }
    if( search ){
        Moves moves = p.generateLegalMoves();
        while( p.makeNextMove(moves) ){
            value = -alphaBeta(p,depth-1,-beta,-alpha);      // Ocenimo pozicijo
            lastMove = p.unmakeMove();
            if( value > alpha ){
                best = value; bestMove = lastMove;
                if( value >= beta ) search = false ;
                else alpha = value;
            }
        }
    }
    // Rezultat iskanja shranimo v transpozicijsko tabelo
    if( best <= oldAlpha ) TTable.record(p.getKey(),0,best,TTable.Upper,0);
    else if( best >= beta )
        TTable.record(p.getKey(),depth,best,TTable.Lower,bestMove);
    else TTable.record(p.getKey(),depth,best,TTable.Exact,bestMove);
    return best;
}
```

Algoritem 8: Alfa–beta s transpozicijsko tabelo

1.9 Celotni algoritem

Vse algoritme opisane v prejšnjih podpoglavjih smo združili v algoritem za igranje šaha. Ta algoritem prikazujejo naslednji algoritmi:

```
int aspirationSearch(Position p){
    for( int depth=1;;depth++ ){
        value = alphaBeta(p,depth,alpha,beta);
        if( timeOut() ) break ; // Če je čas potekel, iskanje končamo
        // Če je rezultat izven isklanega okna, iskanje ponovimo
        ...
        // Določimo novo iskalno okno
        ...
    }
    return value;
}
```

```
int alphaBeta(Position p, int depth, int alpha, int beta){
    // Pogledamo v transpozicijsko tabelo
    TTMove ttMove= TTable.find(p.getKey());
    if( ttMove.Depth >= depth ){
        ...
    }
    // Pogoj za končanje rekurzije
    if( p.endGame() || depth <= 0 ) return quiescenSearch(p,alpha,beta);
    // Klestenje z ničelno potezo
    if( !p.inCheck() && depth > 1 ){
        ...
    }
    // Iskanje najprej nadaljijemo s potezo transpozicijske tabele
    if( ttMove.depth > 0 ){
        ...
    }
    // Alfa-beta iskanje
    Moves moves = p.generateLegalMoves();
    while( p.makeNextMove(moves) ){
        ...
    }
    // Rezultat iskanja shranimo v transpozicijsko tabelo
    ...
    return best;
}
```

```
int quiescentSearch(Position p, int alpha, int beta){
    // V primeru šaha preiskovanje nadaljujemo z globino 1.
    if( p.inCheck() ) return alphaBeta(p,1,alpha,beta);
    // Ocenimo pozicijo in ukrepamo po principu alfa-beta algoritma
    int value = p.evaluate();
    if( value > alpha ){
        if( value >= beta ) return value;
        alpha = value;
    }
    // Ustvarimo seznam vseh možnih jemanj in promocij
    Moves moves = p.generateAllCaptures();
    // Poiščemo najboljšo potezo
    while( p.makeNextMove(moves) ){
        value = - quiescentSearch(p,-beta,-alpha);
        p.unmakeMove();
        if( value > alpha ){
            if( value >= beta ) return value;
            alpha = value;
        }
    }
    return alpha;
}
```

2. Ocenitvena funkcija

Ocenitvena funkcija (Evaluation Function) je najpomembnejši del šahovskih programov. Njena naloga je, da podaja statične ocene pozicij in predstavlja znanje, ki ga vsebujejo šahovski programi. Znanje, ki ga vsebuje ocenitvena funkcija, lahko predstavimo v obliki števila, ki predstavlja verjetnost, da bomo v igri zmagali. Ponavadi se uporabljajo števila, ki nimajo določenega pomena. Oceno pozicije lahko merimo za igralca na potezi ali za njegovega nasprotnika. Če je ocena za enega izmed igralcev dobra, je posledično za drugega slaba (ničelna vsota). Taka ocena je potrebna za pravilno delovanje iskalnih algoritmov. Tako s pomočjo ocenitvene funkcije, dobimo neko število, ki pove za katerega igralca in za koliko je boljša pozicija, glede na njegovega nasprotnika.

Glede na količino znanja, ki jih ocenitvene funkcije vsebujejo, jih delimo na bolj oz. manj kompleksne. Bolj kompleksna je funkcija, več znanja odkriva in časovno je bolj zahtevna. Tako glede na razmerje med hitrostjo in količino znanja, lahko ocenimo moč programa. Če imamo hiter program z malo znanja, bo zmogljivost programa relativno slaba. Tak program lahko izboljšamo z dodajanjem znanja ocenitveni funkciji. Toda če programu dodamo prevelike količine znanja, iskalne algoritme znatno upočasnimo in dobimo program, ki bo pravtako slab. Za optimalno moč programa je tako potrebno uravnavanje njegove hitrosti in znanja. Ta točka uravnavanja je odvisna tudi od tipa nasprotnika. V primeru ko igramo proti drugemu programu, je boljše imeti hitrejši program z malo manj znanja. Tako lahko dosežemo večjo globino iskanja in izberemo boljšo potezo. Kadar pa igramo proti človeku, je bolje, če imamo več znanja. Kajti človek glede na svoje izkušnje ima določeno znanje, s pomočjo katerega zna dobro izkoriščati luknje v znanju programa.

Ocenitvena funkcija, ki smo jo implementirali temelji na ocenitveni funkciji, ki jo je zasnoval Shannon in prikazuje jo naslednja enačba:

$$Ocenitev = \sum_{tip=0}^{6} W[tip](N[tip][bela] - N[tip][\text{črna}])$$

V enačbi N prestavlja število figur določenega tipa in barve in W utež določenega tipa figur. Rezultat te enačbe predstavlja material oz. vsoto razlik vrednosti belih in črnih figur. V ocenitveni funkciji smo uporabili naslednje uteži:

W[kmet]	= 100	W[trdnjava]	= 500
W[skakač]	= 300	W[dama]	= 900
W[tekač]	= 330		

V ocenitveno funkcijo smo vključili še naslednja znanja:

- mobilnost,
  Mobilnost predstavlja število možnih potez, ki jih lahko naredi igralec. Za igralca je bolje, če ima na razpolago več možnih potez kot njegov nasprotnik.

- zaščito kralja in
  S pomočjo zaščite kralja ugotavljamo, kako je zaščiten kralj glede na položaj kmetov v njegovi okolici in figur, ki posredno ali neposredno ogrožajo kralja.

- strukturo kmetov.
  Struktura kmetov ugotavlja, kako so razporejeni kmeti. Ena struktura kmetov je lahko boljša v primerjavi z drugo. Npr., dva kmeta drug za drugim predstavljata slabo strukturo. V kasnejši fazi igre ali končnici lahko ta kmeta postaneta zelo lahek plen za nasprotnika. Iskalni algoritem je omejen z globino iskanja in tako ne more doseči končnice igre ter se tako lahko odloči za slabo potezo.

Implementirana ocenitvena funkcija je relativno preprosta in hitra. Skupaj z iskalnim algoritmom in hitro predstavitvijo igre, smo dobili program katerega zmogljivost je približno 2100 ELO. To potrjujejo rezultati testiranja, ki so predstavljeni v naslednjem poglavju.

3. Analiza algoritma

Algoritme opisane v prvem poglavju smo združili v eno celoto. Tako dobljeni algoritem smo implementirali v programskem jeziku C++. Program je uporabljal alfabeta iskanje, transpozicijsko tabelo, iterativno poglabljanje, aspiracijsko iskanje, iskanje mirovanja, klestenje z ničelno potezo in preprosto ocenitveno funkcijo (material in mobilnost). S pomočjo tega programa smo analizirali časovno in prostorsko zahtevnost algoritma ter njegovo relativno moč. Algoritem smo testirali na računalniku s procesorjem AMD Opteron 848 in Fedora Core release 3 (Heidelberg) operacijskem sistemu. Meritve smo izvajali s pomočjo reševanja 30 pozicij, ki jih vsebuje testna knjižnica "BT2630".

3.1 Časovna zahtevnost

Pričakovana časovna zahtevnost iskalnega algoritma je eksponentna:

$$T(n) = c \cdot 2^{e \cdot n}$$

S pomočjo statističnih meritev lahko na naslednji način ugotovimo eksponentno časovno zahtevnost:

$$T(n_1) = c \cdot 2^{e \cdot n_1}$$
$$T(n_2) = c \cdot 2^{e \cdot n_2}$$

$$\frac{T(n_1)}{T(n_2)} = \frac{2^{e \cdot n_1}}{2^{e \cdot n_2}}$$
$$\frac{T(n_1)}{T(n_2)} = 2^{e \cdot (n_1 - n_2)}$$
$$ln\left(\frac{T(n_1)}{T(n_2)}\right) = e \cdot (n_1 - n_2) \cdot ln(2)$$

$$e = \frac{ln\left(\frac{T(n_1)}{T(n_2)}\right)}{(n_1 - n_2) \cdot ln(2)}$$
$$c = \frac{T(n_1)}{2^{e \cdot n_1}} = \frac{T(n_2)}{2^{e \cdot n_2}}$$

Rezultati meritev:

Globina iskanja	**Čas (sek.)**
1	0.0033
2	0.0058
3	0.0110
4	0.0472
5	0.1243
6	0.3657
7	1.0179
8	2.7404
9	8.9882
10	23.7334
11	83.0451

Izračunana zahtevnost:

$$e = \frac{ln\left(\frac{T(n_{10})}{T(n_{11})}\right)}{(n_{10} - n_{11}) \cdot ln(2)} = \frac{ln\left(\frac{23.7334}{83.0451}\right)}{(10 - 11) \cdot ln(2)} = 1.807$$

$$c = \frac{T(n_{11})}{2^{e \cdot n_{11}}} = \frac{83.0451}{2^{1.807 \cdot 11}} = 8.624 \cdot 10^{-5}$$

Dobljena časovna zahtevnost:

$$T(n) = 8.624 \cdot 10^{-5} \cdot 2^{1.807 \cdot n}$$

3.2 Prostorska zahtevnost

Pričakovana prostorska zahtevnost iskalnega algoritma je eksponentna:

$$S(n) = c \cdot 2^{e \cdot n}$$

Rezultati meritev:

Globina iskanja	**Prostor (MB)**
1	0.000
2	0.000
3	0.000
4	0.008
5	0.042
6	0.375
7	1.254
8	3.532
9	12.424
10	27.639
11	74.796

Izračunana zahtevnost:

$$e = \frac{ln\left(\frac{S(n_{10})}{S(n_{11})}\right)}{(n_{10} - n_{11}) \cdot ln(2)} = \frac{ln\left(\frac{27.6396}{74.796}\right)}{(10 - 11) \cdot ln(2)} = 1.436$$

$$c = \frac{S(n_{11})}{2^{e \cdot n_{11}}} = \frac{74.796}{2^{1.436 \cdot 11}} = 0.00131$$

Dobljena prostorska zahtevnost:

$$S(n) = 0.00131 \cdot 2^{1.436 \cdot n}$$

3.3 Relativna moč programa

Poleg časovne in prostorske zahtevnosti smo ugotavljali še relativno moč programa. Program smo testirali s pomočjo javno dostopnih testnih knjižnic [10], internetnim šahovskim strežnikom [9] (Internet Chess Server) in v igri nasproti šahovskim programom in šahovskim igralcem. Teste smo izvajali na računalniku s procesorjem AMD Opteron 848 in operacijskim sistemom Linux.

Rezultati testov so prikazani v spodnji tabeli. Uporabili smo 8 različnih knjižnic. Prve tri knjižnice nam omogočajo izračun relativne moči šahovskega programa (ELO). Tako smo na osnovi teh knjižnic izračunali, da je moč programa od 2225 do 2451 ELO. Povprečna hitrost nam prikazuje, da je program zelo hiter. Hitrost programa je med 2,500,000 do 3,900,000 vozlišč na sekundo. Zmogljivost programa v primeru knjižnice "Encyclopedia of Chess Middlegames" je nekoliko slabša. Naše mnenje je, da je razlog temu premalo šahovskega znanja, ki ga vsebuje ocenitvena funkcija. Če bi ocenitveni funkciji dodali znanje, bi se verjetno tudi zmogljivost programa pri tej knjižnici povečala.

Knjižnica	**Čas**	**TT**	**%**	**Vozlišč**	**Hitrost**	**w**	**d**	**Rezultat**
BS-2830	900	256	96	2.68e+09	3.90e+06	4.22	15.1	7/27
BT-2630	900	256	94	2.73e+09	3.86e+06	3.82	16.2	18/30
LCT-II	600	256	90	1.68e+09	3.69e+06	4.01	15.3	19/35
BK	60	256	22	1.50e+08	3.20e+06	5.72	10.8	14/24
ECM	20	32	50	5.50e+07	3.56e+06	5.68	10.2	431/879
WAC	5	32	15	1.12e+07	3.21e+06	5.99	9.1	260/300
BWC	5	32	3	2.31e+06	2.49e+06	24.76	4.6	867/1001
WCS	5	32	16	1.20e+07	3.24e+06	5.88	9.2	770/1001

Rezultat - Uspešnost programa TT - Velikost transpozicijske tabele v MB

% - Zasedenost transpozicijske tabele Vozlišč - Povprečno število preiskanih vozlišč na pozicijo

W - Povprečni vejitveni faktor Hitrost - Povprečno število preiskanih vozlišč na sekundo

D - Povprečna globina iskanja Čas - Dovoljeno število sekund za reševanje problema

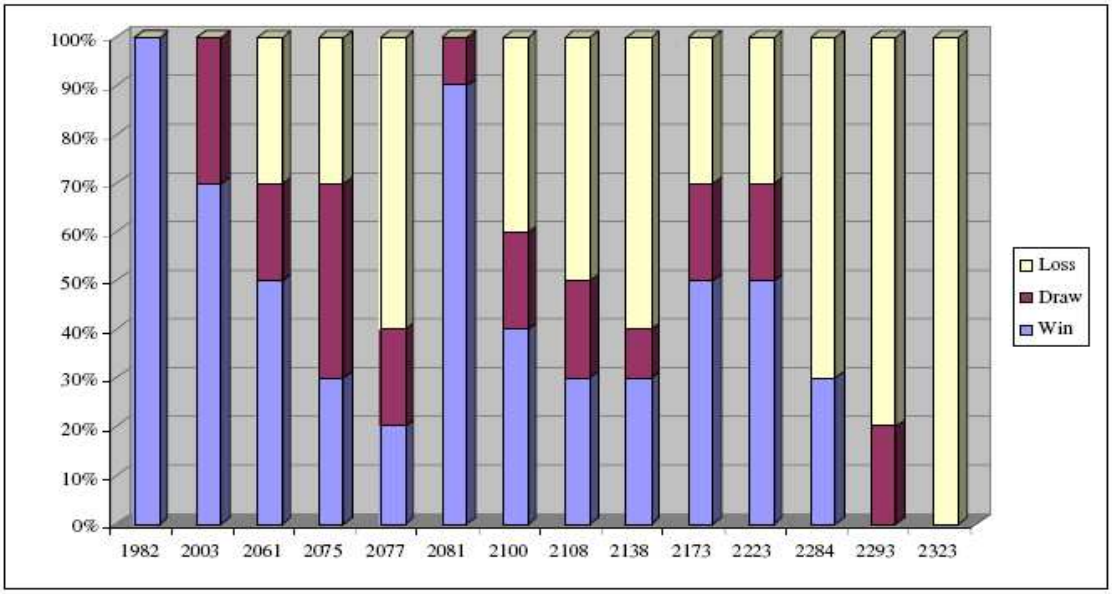
WCS - 1001 Winning Chess Sacrifices; BWC - 1001 Brilliant Ways to Checkmate; WAC - Win at Chess

BK - Bratko–Kopec; ECM - Encyclopedia of Chess Middlegames

Internetni šahovski strežnik nam je omogočil, da je naš program lahko igral nasproti drugim programom in šahovskim igralcem iz celega sveta. Internetni šahovski strežnik je ocenil moč našega programa na 2105 blitz ICC rating v 218 igrah in 2026 standard ICC rating v 35 igrah.

Program smo testirali tudi nasproti šahovskemu igralcu z 2088 ELO. Odigrala sta štiri igre (5 minut na igro) in v vseh igrah je zmagal naš program. Toda nasproti igralcu z 2350 ELO je računalnik bil poražen v dveh igrah (15 minut na igro). Oba igralca sta uporabila normalno strategijo igranja. 5 minutne igre so pokazale, da je kombinatorična moč prevladala nad znanjem šahovskega igralca. Popolnoma nasprotno pa sta se iztekli 15 minutni igri. Tukaj je šahovski igralec zelo učinkovito izkoriščal luknje v znanju ocenitvene funkcije.

Program smo testirali še v igri nasproti komercialnemu šahovskemu programu Chessmaster 8000. Le ta nam je omogočil igranje proti programom z različno relativno močjo. Rezultate teh iger prikazuje slika 3.1, kjer je naš program odigral 10 iger (5 minut na igro) nasproti različnim programom.



Slika 3.1: Histogram zmag, porazov in remijev našega programa nasproti programu Chessmaster 8000

Na osnovi vseh meritev lahko zaključimo, da je relativna moč programa približno 2100 ELO, kar je v primerjavi s svetovnim šahovskim prvakom (2700 ELO) zelo dober rezultat.

Literatura

[1] Ernst A. Heinz: Scalable Search in Computer Chess (Algorithmic Enhancements and Experiments at High Search Depths). Friedr. Viewg & Sohn Verlagsgesellschaft mbH, December 1999.

[2] Aske Plaat: RESEARCH RE:SEARCH & RE–SEARCH, PhD thesis, Erasmus University, 1996.

[3] Yngvi Biörnsson: Selective Depth–First Game Tree–Search, University of Alberta, PhD thesis, Edmonton, Spring 2002.

[4] Breuker D.M., Uiterwijk J.W.H.M. and Herik H.J. van den, Replacement Schemes for Transposition Tables, ICCA Journal, Vol. 17, No.4, pp. 183-193, 1994.

[5] Mark Gordon Brockington: Asynchrounous Parallel Game–Tree Search, University of Alberta, PhD thesis, Edmonton, Spring 1998.

[6] Chess program Gerbil,
dostopno na naslovu http://www.seanet.com/ brucemo/gerbil/gerbil.htm

[7] Ivan Bratko: Prolog in umetna inteligenca, Društvo matematikov, fizikov in astronomov SR Slovenije, Zveza organizacij za tehnično kulturo Slovenije, Ljubljana, 1989.

[8] Radoslav Brglez: Igranje šaha z računalnikom, Univerza v Mariboru, Tehniška fakulteta, diplomsko delo, Maribor, junij 1991.

[9] Internetni šahovski strežnik s preko 150000 registriranimi uporabniki, http://-freechess.org, 2005.

[10] Zbirka testnih knjižnic, http://www.geocities.com/CapeCanaveral/Launchpad/26-40/pgn/tests/, 2005.