UNIVERZA V MARIBORU

Fakulteta za elektrotehniko, računalništvo in informatiko

DIPLOMSKO DELO

Maribor, april 2004 **Borko Bošković**

Univerza v Mariboru
Fakulteta za elektrotehniko,
računalništvo in informatiko

Diplomsko delo univerzitetnega študija

Implementacija računalniškega šaha

Avtor:	Borko Bošković, dipl. inž. rač. in inf.
Študijski program:	Univerzitetni, Računalništvo in informatika
Smer:	Programska oprema
Mentor:	doc. dr. Janez Brest, univ. dipl. inž. rač. in inf.
Komentor:	red. prof. dr. Viljem Žumer, univ. dipl. inž. el.

Sklep o diplomski nalogi

Avtor:	Borko Bošković, dipl. inž. rač. in inf.
Naslov:	Implementacija računalniškega šaha
UDK:	004.8:794.1(043.2)
Ključne besede:	šahovski program, algoritmi, igranje iger, predstavitev šahovske igre, implementacija.
Število izvodov:	4

Zahvala

Zahvaljujem se mentorju doc. dr. Janezu Brestu in komentorju red. prof. dr. Viljemu Žumerju za strokovno pomoč in napotke pri izdelavi diplomskega dela. Zahvaljujem se tudi vsem sodelavcem laboratorija za računalniške arhitekture in jezike.

Še posebej pa se zahvaljujem staršem, ki so mi omogočili študij in zaročenki Sonji, ki me je vzpodbujala pri mojem delu.

UDK: 004.8:794.1(043.2)
Ključne besede: šahovski program, algoritmi, igranje iger, predstavitev šahovske igre, implementacija.

Implementacija računalniškega šaha

Povzetek

V diplomskem delu predstavljamo načrtovanje in implementacijo šahovskega programa. Ta predstavitev vključuje kratko zgodovino računalniškega šaha, različne predstavitve šahovske igre v računalniku in različne iskalne algoritme za igranje iger med dvema igralcema s popolno informacijo in ničelno vsoto. Predstavljene iskalne algoritme in predstavitve igre smo tudi preizkusili. V ta namen smo načrtovali in implementirali šahovski program. Tako v diplomskem delu predstavljamo še načrtovanje in implementacijo ter zmogljivost implementiranega šahovskega programa. Program smo načrtovali tako, da omogoča enostavno dodajanje in testiranje različnih predstavitev igre in iskalnih algoritmov. V okviru implementacijo smo implementirali bitno predstavitev igre, transpozicijsko tabelo, UCI (Universal Chess Interface) vmesnik in naslednje iskalne algoritme: alfa-beta, aspiracijsko iskanje, iskanje na osnovi glavne variante, MTD(f), zbiranje glavne variante in klestenje z ničelno potezo.

UDK: 004.8:794.1(043.2)
Keywords: chess program, algorithms, game playing, chess game representation, implementation.

Computer chess implementation

Abstract

This theses presents a design and implementation of a chess program. The presentation includes a short history of computer chess, representation of chess game in computer and search algorithms which are based on two-player zero-sum game with perfect information. We also tested this representation and search algorithms. For this purpose we designed and implemented chess program. So in theses we also present designing, implementation and efficiency of this chess program. Design of program makes possible a simple adding and testing different representation of game and search algorithms. We implemented bitboard representation, transposition table, universal chess interface and next search algorithms: alpha-beta, aspiration search, principal variation search, MTD(f), collecting the principal variation and null move pruning.

Kazalo

1 Uvod **1**

2 Zgodovina računalniškega šaha **4**

2.1 Prvi šahovski stroj 4
2.2 Turingov "papirnati stroj" 4
2.3 Shannonove strategije 5
2.4 Šah namesto atomskih bomb 5
2.5 Šah in matematika 6
2.6 Algoritem alfa–beta 7
2.7 Računalnik Belle 7
2.8 Šahovski čipi 8
2.9 Osebni računalniki 9
2.10 Napad z obeh strani 10

3 Predstavitev igre **11**

3.1 Predstavitev potez 11
3.2 Predstavitev pozicij 12
3.3 Predstavitev s poljem 8x8 kvadratov 12
3.4 Predstavitev s poljem 10x10 kvadratov 14
3.5 Predstavitev 0x88 14
3.6 Bitna predstavitev 14
3.7 Vračanje potez 17
3.8 Ključi pozicij 17

4 Iskalni algoritmi **19**
4.1 Algoritem MINIMAX 19
4.2 Algoritem NEGAMAX 21
4.3 Algoritem alfa–beta 22
4.4 Iterativno poglabljanje 25
4.5 Aspiracijsko iskanje 25
4.6 Iskanje na osnovi glavne variante 27
4.7 Algoritem MTD(f) 29
4.8 Zbiranje glavne variante 30
4.9 Iskanje mirovanja 32
4.10 Transpozicijska tabela 33
4.11 Zgodovinska hevristika 36
4.12 Klestenje z ničelno potezo 38
4.13 Verifikacija klestenja z ničelno potezo 41
4.14 Dodatne izboljšave 41

5 Generator potez **45**

6 Ocenitvena funkcija **48**
6.1 Izrazi ocenitvene funkcije 48
6.2 Informacije v ocenitveni funkciji 49
6.3 Uglaševanje ocenitvene funkcije 50

7 Implementacija **53**
7.1 Načrtovanje 53
7.2 Implementacija 54
7.3 Grafični uporabniški vmesniki 57
7.3.1 Arena 57
7.3.2 Jose 58
7.4 Testiranje 60

8 Zaključek **62**

Slike

2.1 Prvi šahovski stroj 4
2.2 MANIAC I 6
2.3 Belle 8
2.4 Moč igranja računalnikov glede na globino iskanja 9

4.1 Iskalno drevo algoritma MINIMAX 21
4.2 Vozlišče alfa–beta algoritma 23
4.3 Iskalno drevo algoritma alfa-beta 24

7.1 Razredni diagram 56
7.2 Arena 58
7.3 Jose 2D 59
7.4 Jose 3D 59

Algoritmi

1 Izračun materiala pozicije v predstavitvi s poljem 8x8 kvadratov 13
2 Generator potez za predstavitev s poljem 8x8 kvadratov 13
3 Število nastavljenih bitov v podatkovnem tipu long 15
4 MINIMAX 20
5 NEGAMAX 22
6 Alfa–beta 24
7 Iterativno poglabljanje 25
8 Aspiracijsko iskanje 26
9 NegaScout 28
10 MTD(f) 29
11 Iskanje z minimalnim oknom 30
12 Zbiranje glavne variante 31
13 Iskanje mirovanja 33
14 Alfa–beta z transpozicijsko tabelo 37
15 Klestenje z ničelno potezo 39
16 Verifikacija klestenje z ničelno potezo 42

Uporabljeni simboli in kratice

UCI *Universal Chess Interface.* Standarden vmesnik za povezovanje šahovskih programov in grafičnih uporabniških vmesnikov.

MVV/LVA *Most Valuable Victim/ Least Valuable Attecker.* Metoda za generiranje zaporedja potez.

1. Uvod

Šah že stoletja velja za igro inteligence. Tudi danes je tako, šah predstavlja tekmovanje v inteligenci. Razlog temu je ogromno število kombinacij in kompleksnost, ki jo ta igra vsebuje. Zato je prevladovalo mnenje, da računalnik nikoli ne bo dostojen nasprotnik človeku. Tako so računalniški strokovnjaki na področju umetne inteligence (Artificial Inteligence) dobili izziv in začeli razvijati sisteme, ki bi inteligentno igrali šah.

Z razvojem računalništva so se razvijali tudi računalniški sistemi za igranje šaha. Ti sistemi so postali enakovredni človeku, celo svetovni šahovski prvak jih s težavo premaguje. Kako uspeva računalniku tako briliantno igranje šaha? To je vprašanje, na katerega bomo odgovorili v diplomskem delu. V ta namen smo implementirali šahovski program. Le ta nam je omogočil preizkušanje algoritmov in mehanizmov, ki se uporabljajo v računalniškem šahu.

Diplomsko delo se sestoji iz osem poglavij. Drugo poglavje predstavlja kratko zgodovino računalniškega šaha. V tem poglavju predstavimo prvi šahovski stroj, zahtevnost šahovske igre z računalniškega stališča, pomembnejše računalniške sisteme za igranje šaha, vpliv globine iskanja šahovskih programov na njihovo moč in osnovne sestavne dele šahovskih programov.

V tretjem poglavju opisujemo, kako predstaviti šahovsko igro v računalniku. Opisane so najbolj pogoste predstavitve. Najbolj naravna predstavitev je preslikava šahovske deske v dvodimenzionalno polje 8x8 kvadratov. Za pridobivanje dodatnih informacij o šahovski deski (npr. ali je figura na deski), je potrebno predstavitev zasnovati nekoliko drugače. Tako dobimo predstavitev s poljem 10x10 kvadratov in predstavitev 0x88. Kot najhitrejša in najkompleksnejša predstavitev pa se izkaže bitna predstavitev.

V četrtem poglavju predstavimo iskalne algoritme za igranje iger med dvema nasprotnikoma s popolno informacijo in ničelno vsoto. Najprej predstavimo algoritma MINIMAX in NEGAMAX. Algoritem NEGAMAX predstavlja izboljšavo algoritma MINI-

MAX. Izboljšava temelji na načinu ocenjevanja pozicij in omogoča lažje vzdrževanje kode ter manj napak pomnilniških strani. Naslednjo izboljšavo predstavlja algoritem alfa–beta. Izboljšava tega algoritma temelji na tem, da za določeno pozicijo za katero vemo, da je slabša od trenutno najboljše izbrane, ne računamo za koliko je slabša. Na ta način iskalni algoritem znatno pohitrimo.

Nato predstavimo algoritma, ki omogočata časovno omejeno iskanje. Osnovni algoritem se imenuje iterativno poglabljanje. Izboljšavo tega algoritma pa predstavlja aspiracijsko iskanje. V nadaljevanju poglavja predstavimo še iskanje z minimalnim oknom in algoritma, ki temeljita na tem iskanju. Iskanje z minimalnim oknom je preiskovanje z najmanjšim možnim iskalnim oknom. Algoritma, ki temeljita na tem iskanju, se imenujeta NegaScout in MTD(f).

Poglavje nadaljujemo s predstavitvijo algoritma za zbiranje glavne variante. Ta algoritem kot rezultat iskanja poleg ocenitve, podaja še glavno varianto oz. seznam izbranih potez. Naslednji algoritem, ki ga predstavimo, se imenuje iskanje mirovanja. Ocenitvena funkcija omogoča ocenitev statičnih pozicij. Za ocenjevanje dinamičnih pozicij pa se uporablja iskanje mirovanja.

Zmogljivost algoritmov, ki temeljijo na alfa-beta algoritmu, je zelo odvisna od kvalitete zaporadje generiranih potez. Kvaliteto zaporedja potez lahko izboljšamo s pomočjo generatorja potez, ki temelji na MVV/LVA (Most Valuable Victim/ Least Valuable Attecker) zaporedju. Dodatno pa lahko uporabimo še transpozicijsko tabelo, ubijalsko hevristiko in zgodovinsko hevristiko, ki jih predstavljamo v nadaljevanju četrtega poglavja.

Na koncu četrtega poglavja predstavimo še iskanje z ničelno potezo. To iskanje temelji na ničelni oz. prazni potezi. Ničelna poteza je poteza, ki samo zamenja igralca na potezi. Tako s pomočjo te poteze algoritem ugotavlja, kakšna je pozicija za nasprotnika. To iskanje znatno reducira vejitveni faktor iskalnega drevesa, hkrati pa določene pozicije spregleda. Z omejevanjem izvajanja ničelne poteze ter preverjanjem pravilnosti dobljenega rezultata, lahko število spregledanih pomembnih pozicij zmanjšamo.

V petem poglavju predstavimo še možne implementacije generatorjev potez. Njihova naloga je ustvariti dobro zaporedje vseh možnih potez za poljubno pozicijo v čim krajšem času. Tako ugotovimo, da hiter generator potez potrebuje dobro načrtovano predstavitev igre. Za dobro zaporedje potez lahko uporabimo MVV/LVA shemo, transpozicijsko tabelo ter ubijalsko in zgodovinsko hevristiko. Tako, s pomočjo dobrega generatorja potez, lahko z relativno slabim algoritmom, dosežemo zelo dobre rezultate iskanja.

Zelo pomembno vlogo pri računalniškem šahu ima tudi ocenitvena funkcija. Ta funkcija podaja statično oceno pozicij in predstavlja znanje, ki ga vsebujejo šahovski programi. Tako v šestem poglavju predstavljamo ocenitveno funkcijo ter izraze in informacije, ki jih mora vsebovati. Na osnovi vsebovanega znanja, lahko ocenitvene funkcije klasificiramo na manj in bolj kompleksne. Bolj kompleksne funkcije vsebujejo več znanja, so pa časovno bolj zahtevne. Tako moramo pri načrtovanju in implementaciji ocenitvene funkcije najti najboljše razmerje med njeno hitrostjo in količino vsebovanega znanja. Za izboljšanje kvalitete ocenitvene funkcije lahko, uporabimo še različne metode strojnega učenja. Tako lahko parametre v ocenitveni funkciji prilagodimo programu. Drugi način pa predstavlja zamenjava ocenitvene funkcijer z nevronsko mrežo, ki jo pravtako učimo s pomočjo metod strojnega učenja.

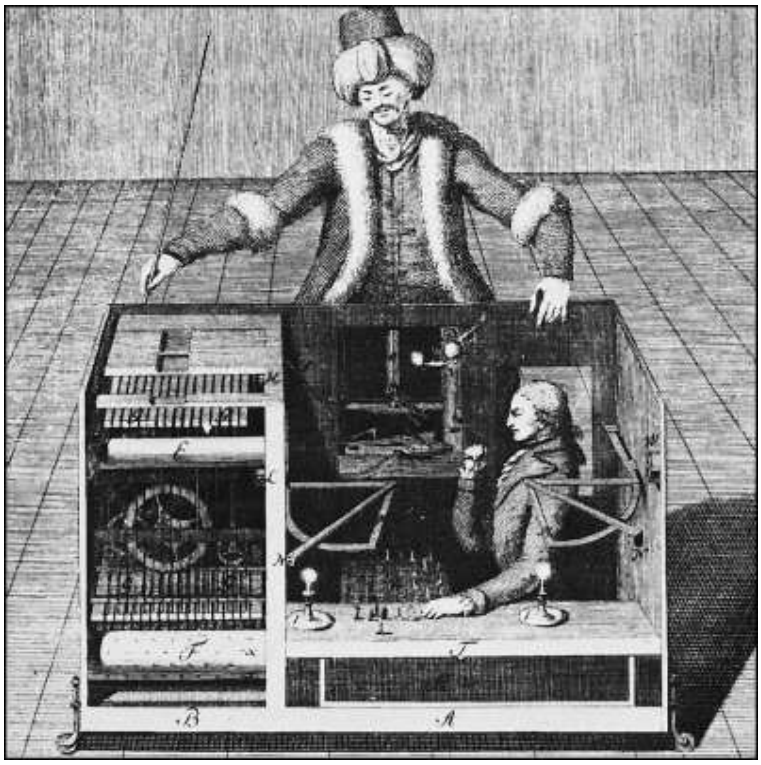
Sedmo poglavje vsebuje predstavitev načrtovanja in implementacije referenčnega šahovskega programa. Program je načrtovan tako, da z implementacijo definiranih vmesnikov in abstraktnih razredov lahko povezujemo in testiramo različne predstavitve igre z različnimi iskalnimi algoritmi. V okviru teh vmesnikov smo implementirali bitno predstavitev igre in iskalne algoritme, ki so predstavljeni v četrtem poglavju. Program dodatno vsebuje implementiran UCI (Universal Chess Interface) vmesnik. S pomočjo tega vmesnika lahko program uporabljamo skupaj z različnimi grafičnimi uporabniškimi vmesniki. Program je implementiran v programskem jeziku java ter tako omogoča njegovo izvajanje na različnih platformah. Program smo še testirali. Ugotovili smo, čeprav smo uporabili objektno otientirano načrtovanje in implementacijo ter programski jezik javo, da je relativno zmogljiv.

Na koncu diplomskega dela podajamo še zaključek o opravljenem delu ter mnenje o zahtevnosti implementacije šahovskih programov in njihovi zmogljivosti.

2. Zgodovina računalniškega šaha

2.1 Prvi šahovski stroj

Leta 1769 je madžar Baron Wolfgang von Kempelen zgradil šahovski stroj za zabavo avstrijske kraljice Marie Theresie [9]. To je bil šahovski stroj z značilno turško obliko. Stroj je presenetljivo dobro igral šah. Razlog temu je bila goljufija. Znotraj stroja je bil skrbno skrit šahovski mojster, ki je namesto stroja igral šah.



Slika 2.1: Prvi šahovski stroj

2.2 Turingov "papirnati stroj"

Zanimivo je dejstvo, da je prvi šahovski program zapisan pred iznajdbo računalnikov [9]. Zapisal ga je vizionar, ki je vedel, da bodo nekega dne iznašli računalnike, katere bo mogoče programirati in posledično z njimi tudi igrati šah. To je bil Alan Turing,

eden od tedaj najboljših matematikov. Znan je še po Turingovem stroju kot pionir računalništva in kot vodja skupine, ki je "zlomila" nemške skrivnoste kode. Tako je pripomogel tudi zmagi nad okupatorjem. Kmalu po končani vojni je svoj program zapisal z inštrukcijami razumljivimi stroju. Ta program pa je nato znal igrati šah.

Turingov papirnati stroj – Alick Glennie, Manchester 1952:

1.e4 e5 2.Nc3 Nf6 3.d4 Bb4 4.Nf3 d6 5.Bd2 Nc6 6.d5 Nd4 7.h4 Bg4 8.a4 Nxf3+ 9.gxf3 Bh5 10.Bb5+ c6 11.dxc6 0–0 12.cxb7 Rb8 13.Ba6 Qa5 14.Qe2 Nd7 15.Rg1 Nc5 16.Rg5 Bg6 17. Bb5 Nxb7 18.0–0–0 Nc5 19.Bc6 Rfc8 20. Bd5 Bxc3 21.Bxc3 Qxa4 22.Kd2 Ne6 23.Rg4 Nd4 24.Qd3 Nb5 25.Bb3 Qa6 26.Bc4 Bh5 27.Rg3 Qa4 28.Bxb5 Qxb5 29.Qxd6 Rd8 0–1.

2.3 Shannonove strategije

V enakem obdobju kot Turing je živel še en dober matematik Claude Shannon. On je preučeval, kako računalnik "naučiti" igrati šah [9, 8]. Spoznal je, da šah ima ogromno število kombinacij, ki jih ni mogoče pregledati. Tako je začel preiskovati med "A–strategijami", ki pregledajo vse možne kombinacije in "B–strategijami", ki določene kombinacije izločijo iz preiskovanja. Tako tudi danes razlikujemo med "brute force" in selektivnimi strategijami. Obe sta bolj ali manj uspešni za določene tipe problemov.

2.4 Šah namesto atomskih bomb

Med drugo svetovno vojno so v Los Alamosu, ki se nahaja v ZDA, zgradili ogromni laboratorij. Namen laboratorija je bil razviti atomsko orožje. Madžarsko ameriški matematik John von Neumann je leta 1946 bil zadolžen za izgradnjo močnega računalnika za zahtevne izračune pri verižnih atomskih reakcijah. Leta 1950 so zgradili računalnik imenovan MANIAC I. Vseboval je na tisoče elektronk in preklopnikov. Ta računalnik je izvajal približno 10000 inštrukcij na sekundo. Prav tako ga je bilo mogoče programirati. Namesto začetnega namena so na računalniku začeli izvajati druge eksperimente. Eden od prvih programov je bil šahovski program. Ta program je bil zasnovan tako, da je šahovsko desko omejil na 6x6 polj oz. igral je brez lovcev. Program je v približno 12 minutah preiskal kombinacije do globine štirih potez [9, 8]. Za isto globino z lovci je potreboval 3 ure.

Slika 2.2: MANIAC I

Ta program je v sredini petdesetih odigral tri partije. Najprej je igral proti sebi in zmagal je igralec z belimi figurami. Potem je igral proti močnemu igralcu in po 10 urah je partijo izgubil. Tretjo igro je igral proti ženski, ki se je igro začela učiti teden pred dvobojem. Program je zmagal v 23 potezi. To je bilo prvič v zgodovini, da je človek izgubil proti računalniku v igri intelektualnega tipa.

MANIAC I – Človek, Los Alamos 1956:

1.d3 b4 2.Nf3 d4 3.b3 e4 4.Ne1 a4 5.bxa4 Nxa4 6.Kd2 Nc3 7.Nxc3 bxc3+ 8.Kd1 f4 9.a3 Rb6 10.a4 Ra6 11.a5 Kd5 12.Qa3 Qb5 13.Qa2+ Ke5 14.Rb1 Rxa5 15.Rxb5 Rxa2 16.Rb1 Ra5 17.f3 Ra4 18.fxe4 c4 19.Nf3+ Kd6 20.e5+ Kd5 21.exf6Q Nc5 22.Qf6xd4+ Kc6 23.Nf3–e5 mat.

2.5 Šah in matematika

Glavni problem pri izdelavi šahovskih programov je zelo veliko število različnih kombinacij oz. pozicij [9]. Povprečno število potez v eni poziciji je 35. V dveh potezah tako imamo povprečno 1225 pozicij, v štirih potezah $1.5 \bullet 10^6$ pozicij in po šestih potezah $1.8 \bullet 10^9$ pozicij. Povprečno število potez igralca v igri je 40. Tako je vseh pozicij

približno 10^{123}. Tako veliko število pozicij pa v današnjem času ni mogoče preiskati z nobenim računalnikom ali strojem.

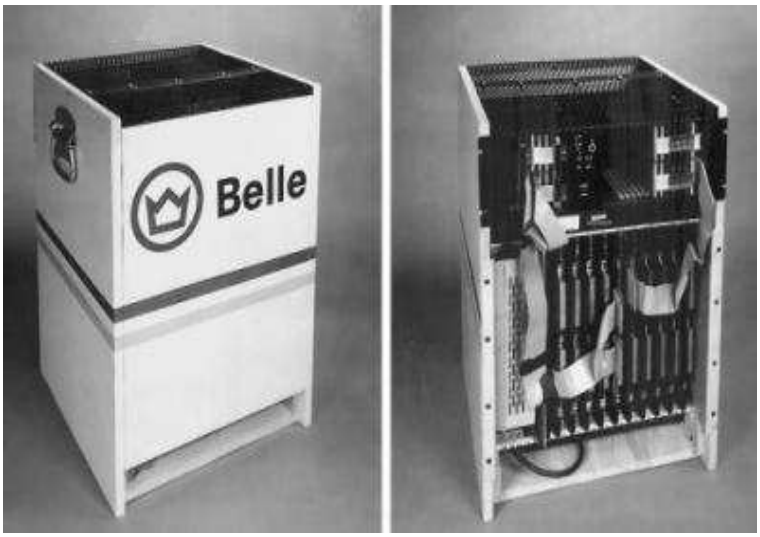
Tudi človek ni idealen igralec. Tako se postavlja naslednje vprašanje. Do katere globine je potrebno preiskovati, da bomo premagali človeško strategijo? Prvi računalniki so generirali in ocenili okoli 500 pozicij na sekundo oz. 90 000 pozicij v treh minutah (dovoljen čas za potezo na turnirju). S to hitrostjo so računalniki preiskovali pozicije do globine treh potez. Njihova igra je bila zelo slaba, na nivoju začetnikov. Sčasoma so računalniki postali hitrejši in njihova hitrost je omogočala preiskati 15 000 in več pozicij na sekundo. Tako se je povečala tudi globina iskanja za 1. Toda tudi ti računalniki so igrali zelo površno.

2.6 Algoritem alfa–beta

Prvo prelomnico v razvoju računalniškega šaha predstavlja leto 1958. Tega leta so trije znanstveniki z univerze Carnegie–Mellon v Pittsburgh–u (Newell, Shaw in Simon) iznašli pomembno odkritje [9, 8]. S pomočjo tega odkritja so znatno reducirali preiskovalno drevo, brez vpliva na končni rezultat. To so dosegli s pomočjo alfa–beta algoritma. To je algoritem, ki temelji na matematiki in ne uporablja nobenega šahovskega znanja. S pomočjo tega algoritma so nekoliko povečali globino iskanja, toda zahtevnost algoritma je bila še vedno eksponentna.

2.7 Računalnik Belle

Ken Thompson je bil računalniški znanstvenik in ni imel milijon dolarjev za računalnik, ki bi bil za 5 do 25 krat hitrejši od navadnih računalnikov. Zato se je s kolegi odločil zgraditi posebej namenski računalnik. Ta računalnik je vseboval nekaj 100 čipov, vrednih okoli 20 000 dolarjev. Računalnik so poimenovali Belle in namenjen je bil samo igranju šaha [9, 8]. Zmožen je bil preiskovati okoli 180 000 pozicij na sekundo. S to hitrostjo je v času za turnirsko igro mojsterske kategorije, dosegel globino iskanja od 8 do 9 potez. Računalnik je zmagal na svetovnem šahovskem prvenstvu in na vseh ostalih računalniških turnirjih od 1980 do 1983 leta. Prvi računalnik, ki ga je premagal, je bil ogromen Cray X–MP. Njegova vrednost pa je bila nekaj tisoč krat večja od računalnika Belle.



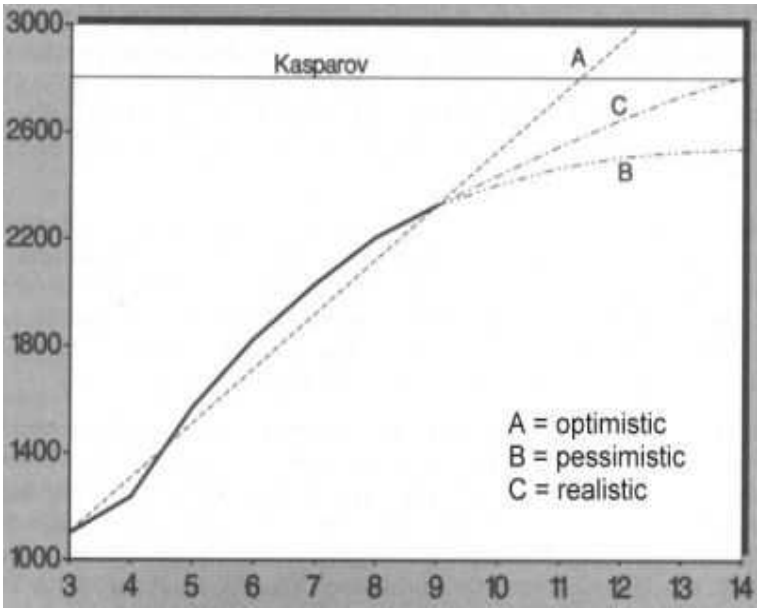
Slika 2.3: Belle

2.8 Šahovski čipi

Prof. Hans Beliner, računalniški znanstvenik na univerzi Carnegie–Mellon je nadaljeval delo, ki ga je začel Ken Thompson. Želel je postati svetovna korespodenca v tekmovanjih računalniškega šaha. Zato je zgradil stroj imenovan HiTech [9]. Skupaj s svojim študentom (Carl Ebeling) sta razvila čip za generiranje potez. S pomočjo 64 čipov, ki so se nahajali v paralelnem HiTech računalniku, je tesno izgubil zmago za svetovnega prvaka leta, 1986 proti računalniku Cray.

Kmalu po tem so Berlinerjevi študentje Feng–hsiung Hsu, Murray Campbell in drugi razvili njihov lasten računalnik imenovan ChipTest in pozneje Deep Thought. Njegova vrednost je bila okoli 5 000 dolarjev in pregledoval je okoli 500 000 pozicij na sekundo. Kasneje sta Hsu in Campbell odšla in se pridružila podjetju IBM. Tukaj se jima je pridružil Joe Hoane in zgradili so računalnik imenovan Deep Blue. Proti temu računalniku je igral tudi svetovni šahovski prvak Garry Kasparov [9, 8]. Računalnik je bil sestavljen iz velikega števila posebej namenskih čipov. Vsak čip je bil zmožen procesirati od 2 do 3 milijone pozicij na sekundo. Z uporabo 200 takih čipov je hitrost programa znašala 200 milijonov pozicij na sekundo.

Kaj predstavlja za računalnik hitrost preiskovanja 200 milijonov pozicij na sekundo? Ken Thompson je v 80–tih letih izvedel zanimiv eksperiment o korelaciji med globino iskanja in močjo igranja [9]. Thompson je pustil igrati Belle–a samega proti sebi z naraščanjem globine iskanja. Z globino iskanja 1, je moč igranja znašala 200 ELO točk. Z globino 4, je moč igranja zrasla na 1230 ELO točk in na globini 9, na 2328 ELO točk.



Slika 2.4: Moč igranja računalnikov glede na globino iskanja

Z nadaljevanjem krivulje je prišel do zaključka, da bi računalniki z globino 14 imeli moč svetovnega šahovskega prvaka (2800 ELO). Da bi računalnik dosegel to globino mora pregledovati bilijon pozicij na sekundo. Tak računalnik bi lahko zmagal na človeškem svetovnem šahovskem prvenstvu. Deep Blue je prišel temu cilju že zelo blizu, toda tega naslova še nima. Tako se postavlja samo še vprašanje časa, kdaj bo računalnik postal boljši igralec šaha v primerjavi z človekom.

2.9 Osebni računalniki

Tudi način programiranja in uporabljene ideje so zelo pomembne pri razvoju šahovskih programov. Danes najboljši programi, kot sta npr. Fritz in Junior zmoreta preiskati okoli milijon pozicij na sekundo. Njihova realna moč je približno 2700 ELO točk.

Z njimi se tako lahko enakopravno pomeri le prvih 100 najboljših igralcev sveta. V hitropoteznem šahu pa se proti njim lahko kosata le dva ali trije igralci sveta.

2.10 Napad z obeh strani

Zelo pomemben del šahovskih programov so otvoritvene knjižnice. Te knjižnice vsebujejo zbrana znanja in izkustva šahovskih mojstrov skozi več generacij. Možno jih je shraniti na disk in jih uporabiti v začetnih fazah igre. Knjižnice vsebujejo na desetine milijonov otvoritvenih pozicij in imajo popolno statistiko o vsaki poziciji. Tako prvih 15 do 20 potez programi odigrajo s pomočjo otvoritvenih knjižnic in ne uporabljajo iskalnih algoritmov [9]. Brez otvoritvenih knjižnic bi programi bili zelo hendikepirani.

Podobno kot otvoritvene knjižnice, programi uporabljajo še podatkovno bazo končnic. S pomočjo te podatkovne baze programi praktično igrajo brez napak v končnicah, ki vsebujejo od 4 do 5 figur. Pionir podatkovne baze končnic je Ken Thompson. Generiral in shranil je vse pozicije končnic s štirimi in petimi figurami. Z uporabo te podatkovne baze, računalnik za vsako od končnic igra brez napak. V vsaki poziciji natančno ve ali pozicija vodi k zmagi, porazu ali remiju in v koliko potezah. Tudi Deep Blue je uporabljal to podatkovno bazo, celo program Fritz jo vsebuje implementirano v iskalnem drevesu. Analizirajmo še podatkovno bazo končnic pozicij, ki vsebujejo 6 figur. Te končnice vsebujejo od 8 do 20 bilijonov pozicij. V nekaterih od končnic je potrebno odigrati do konca igre tudi do 200 natančnih potez. V takih končnicah človek nima nobene možnosti proti računalniku.

Na osnovi velikega števila kombinacij, ki jih vsebuje šahovska igra, lahko ugotovimo, da se podatkovna baza končnic in otvoritvene knjižnice v šahovskih programih nikoli ne bodo srečale. Vedno jih bo povezoval vedno močnejši iskalni algoritem. Tako se postavlja samo še vprašanje časa, kdaj bo računalnik v dvoboju z človekom nepremagljiv.

3. Predstavitev igre

Podobno, kot vsak drugi problem, je tudi šahovsko igro potrebno na nek način predstaviti v računalniku. Predstavitev šaha mora omogočati shranjevanje in manipulacijo s šahovskimi pozicijami in potezami. Pozicijo igre predstavlja trenutno stanje v igri, poteza igralca pa premik ene od njegovih figur v trenutni poziciji. Predstavitev igre je potrebno zasnovati tako, da omogoči izvajanje naslednjih operacij [8]:

- izvajanje določene poteze (v primeru zahtev uporabnika in kot del iskalnega algoritma),
- vračanje poteze (v primeru zahtev uporabnika in kot del iskalnega algoritma),
- predstavitev igre uporabniku,
- ustvarjanje seznama vseh možnih potez in
- ocenjevanje pozicije igre.

Vse naštete operacije razen predstavitve igre, morajo biti hitre, saj se uporabljajo v iskalnih algoritmh.

3.1 Predstavitev potez

Predstavitev potez mora biti kompaktna in jedrnata. Saj jih na eni strani moramo hitro dekodirati in odigrati, na drugi strain pa shranjevati za kasnejšo uporabo. Predstavitev poteze mora vsebovati podatke o figuri poteze (figura, ki se premika), načinu premika, opcijsko podatke o jemani oz. figuri zamenjave in ocenitvi poteze. Ena od možnih predstavitev je predstavitev s pomočjo 32 bitov. Tako s pomočjo 32 bitnega celega števila (tip integer) lahko manipuliramo z potezami in jih shranjujemo za kasnejšo uporabo.

tip	figura 1	polje 1	polje 2	figura 2	ocena poteze
xxx	xxxx	xxxxxx	xxxxxx	xxxx	xxxxxxxxx

Pri tej predstavitvi so prvi trije biti namenjena predstavitvi tipa poteze. Tako poteze klasificiramo v skupine (premik, jemanje, an passant, zamenjava in rokada), kar omogoča njihovo hitrejše dekodiranje. Naslednji štirje biti predstavljajo figuro poteze. Za določitev premika pa se uporablja naslednjih 12 bitov. Prvih šest bitov predstavlja polje, s katerega se figura premika in naslednjih šest bitov, polje na katerega se figura premika. Naslednji štirje biti pa se uporabljajo v primeru potez jemanja in zamenjave. Predstavljajo pa jemano figuro oz. figuro zamenjave. Preostalih devet bitov pa predstavlja oceno poteze. Opisana predstavitev vsebuje tudi naslednjo izjemo: v primeru, da gre za zamenjavo figure, takrat figuro poteze predstavlja figura, ki jo zamenjujemo za kmeta.

Tako ta predstavitev omogoča urejanje potez s pomočjo urejanja enodimenzionalnega polja števil. To lastnost predstavitve lahko zelo učinkovito uporabimo pri generatorju potez. Le ta mora, poleg ustvarjanje seznama potez, poteze tudi urediti glede na njihove ocene. Urejeno zaporedje generiranih potez, pa kot bomo videli v nadaljevanju, zelo vpliva na učinkovitost iskalnih algoritmov. Poleg urejanja, ta predstavitev omogičajo še enostavno shranjevanje potez za kasnejšu uporabo ter hitro dekodiranje s pomočjo bitnih operacij in njihovo izvajanje nad pozicijami.

3.2 Predstavitev pozicij

Poleg potez moramo poznati tudi položaje figur na deski, kot tudi določene nevidne informacije (kdo je na potezi, kateri igralec na katero stran lahko izvaja rokado, kje je možna poteza en passant, kot tudi določene informacije o prejšnjih potezah za odkrivanje treh ponovljenih pozicij). Za pridobivanje naštetih informacij in za izpolnjevanje naštetih zahtev obstaja dosti različnih predstavitev. V nadaljevanju tega poglavja bomo predstavili nekaj najbolj uporabljenih.

3.3 Predstavitev s poljem 8x8 kvadratov

Ta predstavitev je najbolj naravna in preslika šahovsko desko v dvodimenzionalno polje 8x8 kvadratov. Vsak kvadrat ima lahko vrednost ene od dvanajst različnih figur, ali praznega kvadrata. Prednost te predstavitve je njena enostavnost ter enostavno

izračunavanje materiala pozicij [8]. Podrobnejši opis materiala pozicij je predstavljen v podpoglavju 6.2.

```
int material = 0;
for( int i=0; i<8; i++ )
    for( int j=0; j<8; j++ )
        material += board.square[i][j].piece.value();
```

Algoritem 1: Izračun materiala pozicije v predstavitvi s poljem 8x8 kvadratov

Nekoliko bolj zapletena je implementacija generatorja potez oz. generiranje vseh možnih potez za določeno pozicijo. Del generatorja potez, ki obravnava poteze kmetov, prikazuje algoritem 2. Dodatno slabost predstavlja še ugotavljanje ali je igralec v šahu. To ugotavljanje zahteva zapleteno kodo in nepotrebno upočasni algoritme iskanja.

```
for( int i=0; i<8; i++ ){
    for( int j=0; j<8; j++ ){
        switch( board.square[i][j].piece() ){
            case Piece.WHITE_PAWN:
                if( board.square[i+1][j].piece() == Piece.EMPTY ){
                        // Ustvarimo potezo, ki premakne kmeta za eno polje naprej.
                }
                if( i == 2 &&
                    board.square[i+1][j].piece() == Piece.EMPTY &&
                    board.square[i+2][j].piece() == Piece.EMPTY ){
                        // Ustvarimo potezo, ki premakne kmeta za dve polji naprej.
                }
                if(  j > 0 &&
                    board.square[i+1][j-1].pieceColor() == Piece.BLACK ){
                        // Ustvarimo potezo, ki z kmetom jemlje črno figuro.
                }
                if(  j < 7 &&
                    board.square[i+1][j+1].pieceColor() == Piece.BLACK ){
                        // Ustvarimo potezo, ki z kmetom jemlje črno figuro.
                }
                ...
                break ;

            ...
        }
    }
}
```

Algoritem 2: Generator potez za predstavitev s poljem 8x8 kvadratov

3.4 Predstavitev s poljem 10x10 kvadratov

Ta predstavitev vsebuje dvodimenzionalno polje 10x10 kvadratov in predstavlja razširitev prejšnje predstavitve. Z dodatnimi kvadrati pridobimo dodatne informacije (meje deske) in poenostavitve [8]. Tako lahko zmanjšamo število pogojev v stavkih if generatorja potez, posledično pa pohitrimo generator potez in iskalne algoritme.

3.5 Predstavitev 0x88

Naslednja predstavitev se imenuje 0x88. Ime je dobila po testu, ki odkriva meje deske [8]. Deska je predstavljena z enodimenzionalnim poljem velikosti 128 elementov oz. 16x8 kvardati. Taka predstavitev vsebuje uporabljeno (levo) in neuporabljeno (desno) stran deske. Indeksi uporabljene deske so naslednji:

112	113	114	115	116	117	118	119
96	97	98	99	100	101	102	103
80	81	82	83	84	85	86	87
64	65	66	67	68	69	70	71
48	49	50	51	52	53	54	55
32	33	34	35	36	37	38	39
16	17	18	19	20	21	22	23
0	1	2	3	4	5	6	7

Tako glede na indeks polja lahko na hiter način ugotovimo ali je polje znotraj ali izven uporabljene strani deske. Če je v indeksu nastavljen bit 0x80 (šestnajstiško) pomeni, da je polje indeksa izven uporabljene in neuporabljene deske. Če je v indeksu nastavljen bit 0x08 pa pomeni, da je polje izven uporabnega dela deske. Tako s pomočjo združevanja teh dveh bitov dobimo število 0x88. To število nam omogoča, s pomočjo bitne operacije IN (AND), hiter test ali indeks predstavlja polje na uporabljenem delu deske. Ta predstavitev je nekoliko kompleksnejša od prejše, toda omogoča lažje in hitrejše ugotavljanje, ali je figura na deski.

3.6 Bitna predstavitev

Četrta predstavitev je bitna predstavitev. Je kompleksnejša od prejšnjih. Zaradi bitnih operacij pa se izkaže kot hitrejša oz. zmogljivejša. Npr. določeni biti v predstavitvi predstavljajo napadena polja s strani belih kmetov, določeni pa črne figure. Tako s

pomočjo bitne operacije IN med temi biti dobimo bite, ki predstavljajo črne fugure napadene s strani belih kmetov.

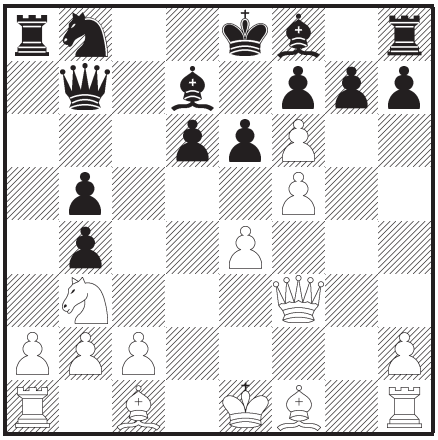
Zmogljivost bitne predstavitve izvira iz učinkovitosti bitnih operacij v računalniku. Tako za učinkovito bitno predstavitev lahko uporabimo naslednja izraza (b & -b) in (b & (b -1)). Prvi izraz, vse bite razen zadnjega, postavi na nič, drugi izraz pa postavi samo zadnji bit na nič. Na žalost pa iskanje poljubnega bita in število nastavljenih bitov pa zahteva nekoliko zahtevnejše algoritme [1, 2]. Npr. število nastavljenih bitov v podatkovnem tipu long prikazuje algoritem 3.

```
int stBitov(long b) {
    int st;
    for( st=0; b! = 0; st++, b &= (b-1) ) ;
    return st;
}
```

Algoritem 3: Število nastavljenih bitov v podatkovnem tipu long

Prikazan algoritem je zelo učinkovit v primeru malega števila nastavljenih bitov. Taki primeri so tudi najbolj pogosti v bitni predstavitvi. Razlog temu je šahovska deska, ki vsebuje 64 polj in maksimalno 32 figur. Tako šahovsko desko predstavimo s pomočjo 64 bitnega podatkovnega tipa, ki v najslabšem primeru ima nastavljenih 32 bitov.

Bitna predstavitev za vsak tip figure vsebuje 64 bitno število. Ker imamo 6 različnih belih in 6 različnih črnih figur, predstavitev vsebuje 12 takih števil. Npr., če imamo nasledno pozicijo:



je bitna predstavitev belih kmetov naslednja:

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Bitno predstavitev vseh črnih figur pa dobimo s pomočjo naslednjega izraza:

$$cFugure = cKmeti|cSkakaca|cLovca|cTrdnjavi|cDama|cKralj. \tag{3.1}$$

Bitna predstavitev nam poleg informacij o trenutni poziciji omogoča tudi druge informacije. Npr. polja, na katera je mogoče premakniti bele kmete, v primeru premika za eno polje naprej, dobimo s pomočjo naslednjega izraza [8]:

$$premikBKmetov = (bKmeti << 8)\& \sim vseFigure \tag{3.2}$$

Podrobneje si poglejmo predstavljen izraz. Najprej predstavitev belih kmetov premaknemo za 8 oz. figure premaknemo za eno polje naprej.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Nato negiramo predstavitev vseh figur in dobimo predstavitev praznih polj.

$$\begin{matrix} 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 1 & 0 \end{matrix}$$

Z bitno operacijo IN nad dobljenima predstavitvama dobimo predstavitev za polja, na katera je možno postaviti bele kmete, kadar jih premikamo za eno polje naprej.

$$\begin{matrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{matrix}$$

Ta primer prikazuje učinkovitost bitne predstavitve. S pomočjo nekaj bitnih operacij smo ugotovili, na katera polja je mogoče odigrati vse bele kmete.

3.7 Vračanje potez

V iskalnih algoritmih in v interakciji z uporabnikom potrebujemo vračanje potez. To lahko naredimo na dva načina. Prvi način shranjuje pozicije na sklad ter jih kasneje, v primeru vračanja potez, jemlje s sklada. Slabost tega načina je njegova počasnost. Drugi način je znatno hitrejši. Predstavitvi pozicij dodaja informacije o odigranih potezah. Tako v primeru vračanja poteze, se na osnovi zadnje odigrane poteze, preoblikuje pozicija. Informacije, ki pa jih moramo dodati so nevidne informacije o rokadi, en passnt potezi in sama poteza.

3.8 Ključi pozicij

Kadar imamo določeno predstavitev pozicije, je potrebno pozicije shranjevati za kasnejšo uporabo, ter jih med seboj tudi primerjati. Možna rešitev tega problema je uporaba 64 bitnih ključev pozicij (Zobrist keys) [10].

Ideja je v tem, da najprej ustvarimo in napolnimo trodimenzionalno polje 64 bitnih števil z naključnimi vrednostmi. Dimenzije v trodimenzionalnem polju imajo naslednji pomen: barvo, tip in položaj figure. Za določitev ključa je najprej potrebno njegovo vrednost nastaviti na 0. Naslednji korak je dodajanje figur ključu. To naredimo s pomočjo izvajanja logične operacije XOR s ključem in števili trodimenzionalnega polja naključnih števil, ki pripadajo figuram na deski. Npr., če imamo na polju e5

belega kmeta, naredimo XOR operacijo s ključem in naključnim številom, ki pripada belemu kmetu in polju e5 (zobrist[beli][kmet][e5]). Nato ključu dodamo še informacijo o igralcu, ki je na potezi. V primeru, da je na potezi črni igralec, z naključnim konstantnim številom in ključem, izvedemo še eno logično operacijo XOR. V nasprotnem primeru ne naredimo ničesar.

Opisani postopek ima še eno lepo lastnost. Na enak način kot smo ključu dodajali figure, jih lahko tudi odstranjujemo. Tako ključev ni potrebno računati za vsako pozicijo posebej. Lahko uporabimo ključ prejšnje pozicije in na osnovi poteze izračunamo nov ključ. Npr. če imamo belega kmeta na polju e5 in ga želimo premakniti na polje e6. Najprej ključu odstranimo figuro na enak način, kot smo jo v prejšnjem primeru dodali. Nato pa ključu odstranjeno figuro še dodamo na polju e6. Tako zgrajeni ključi so sicer lahko za različne pozicije enaki in lahko povzročijo napako v iskalnem algoritmu. Vendar verjetnost ponovitve je tako mala, da jo lahko zanemarimo.

Opisane ključe pozicij lahko uporabimo pri implementaciji transpozicijske tabele (podpoglavje 4.10). S pomočjo te tabele se poskušamo izogniti ocenitvi istih pozicij večkrat. Ključe lahko uporabimo tudi za implementacijo strukture kmetov oz. analizo strukture kmetov pri ocenitveni funkciji (podpoglavje 6.2). Dodatno pa lahko ključe pozicij uporabimo še pri odkrivanju večnega šaha, odkrivanju remi pozicij (treh ponovljenih pozicij) in za kreiranje otvoritvenih knjižnic.

Opisane predstavitve v šahovskih programih uporabljajo iskalni algoritmi. Zato je zmogljivost iskalnih algoritmov kakor tudi šahovskih programov, zelo odvisna od izbrane predstavitve in njene implementacije. Tako je lahko šahovski program z nekoliko slabšim iskalnim algoritmom in dobro predstavitvijo igre, boljši od programa z boljšim iskalnim algoritmom, ki pa ima slabšo predstavitev igre.

4. Iskalni algoritmi

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":

$$eval_w(p) + eval_b(p) = 0. \tag{4.1}$$

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.

4.1 Algoritem MINIMAX

Igralca v šahovski igri izmenjujeta poteze in oba poskušata izbrati najboljšo svojo potezo oz. maksimirati svojo vrednost, posledično pa minimalizirati nasprotnikovo. Tako glede na trenutno pozicijo identificiramo MIN in MAX igralca. V začetni poziciji

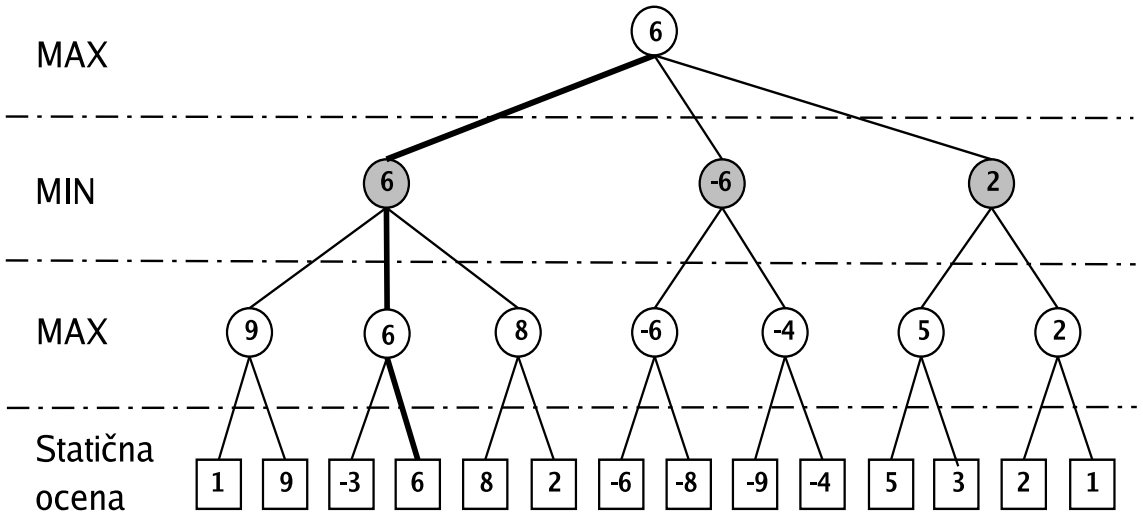
MAX igralca predstavlja beli igralec, ki je na potezi. MIN igralca pa predstavlja črni igralec. Na osnovi opisanega lahko formuliramo algoritem za igranje iger med dvema nasprotnikoma s popolno informacijo in ničelno vsoto. To naredimo tako, da simuliramo obnašanje igralcev MIN in MAX oz. ti. načela MINIMAX.

```
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 4: MINIMAX

Način delovanja prikazanega algoritma oz. njegovo iskalno drevo prikazuje slika 4.1. 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, 3, 7, 4, 12]. Potem 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 4.1.



Slika 4.1: Iskalno drevo algoritma MINIMAX

4.2 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. 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, 7, 4].

```
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;
}
```

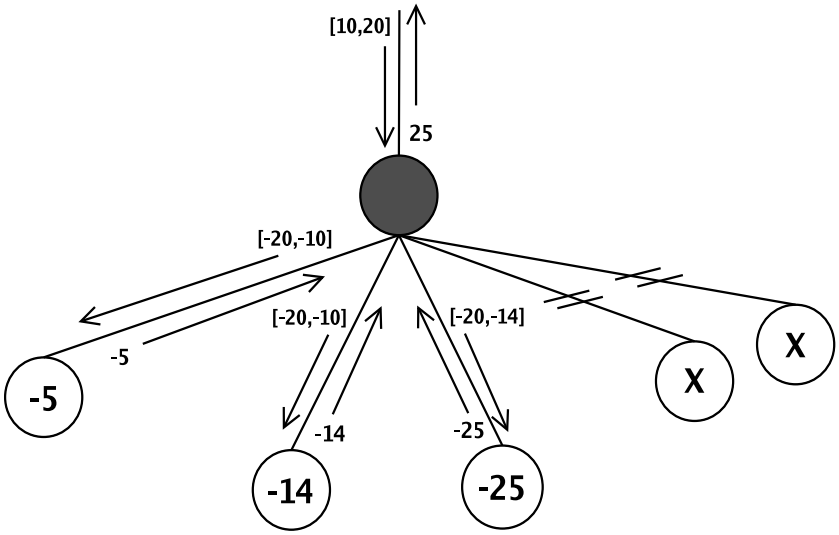
Algoritem 5: NEGAMAX

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 $1.8 \bullet 10^9$ vozlišč ali za globino 10 drevo $2.8 \bullet 10^{15}$ vozlišč.

4.3 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 najslabši scenarij za nasprotnika, ki mu je že zagotovljen [13]. 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.

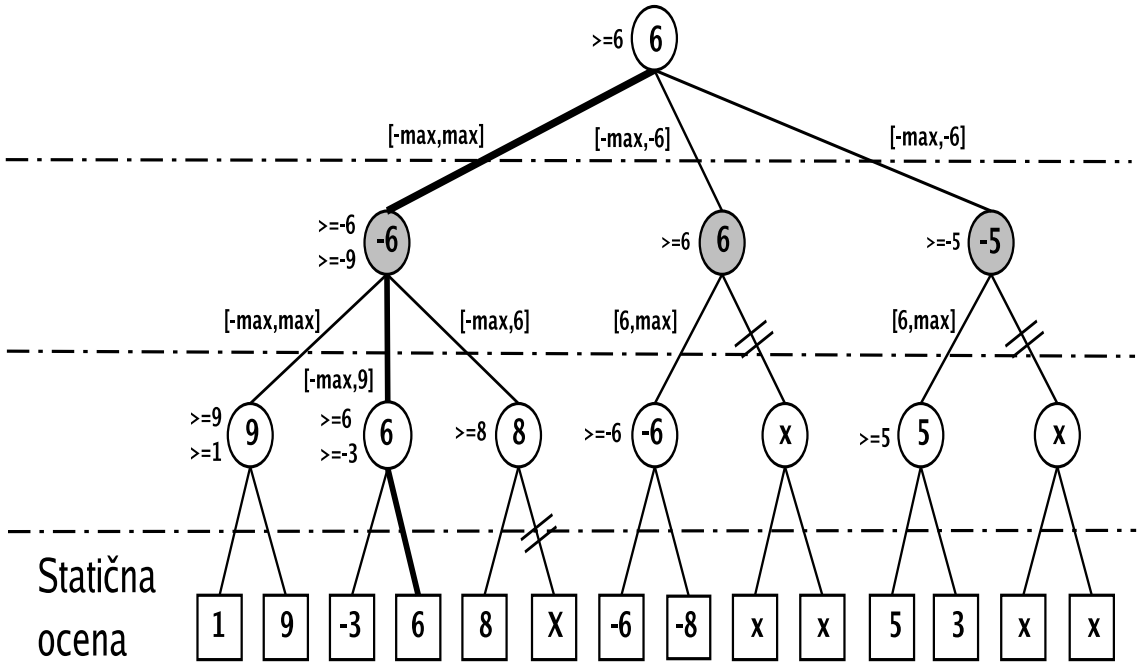
Poglejmo opisan algoritem na primeru vozlišča, ki ga prikazuje slika 4.2. Vozlišče prejme meji alfa in beta oz. iskalno okno (search window) 10,20. Ker algoritem temelji na NEGAMAX algoritmu se preiskovanje nadaljuje z oknom -20,-10. Ocenitev prvega vozlišča je -5. Rezultatu spremenimo predznak in dobimo 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.



Slika 4.2: Vozlišče alfa–beta algoritma

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:

$$w^{\lceil \frac{d}{2} \rceil} + w^{\lfloor \frac{d}{2} \rfloor} - 1 \tag{4.2}$$



Slika 4.3: Iskalno drevo algoritma alfa-beta

```
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 6: 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 6. Primer iskalnega drevesa in način delovanja algoritma pa prikazuje slika 4.3.

4.4 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 7.

```
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 7: Iterativno poglabljanje

Na prvi pogled se zdi, da je to iskanje zelo počasno. Če pa uporabljamo transpozicijsko tabelo, 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 [4].

4.5 Aspiracijsko iskanje

Iterativno poglabljanje uporablja alfa-beta algoritem z zelo velikim oknom. To okno lahko zmanjšamo in spreminjamo glede na preiskovanja v prejšnjih iteracijah. Kajti ver-

jetnost, 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 8.

```
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,i,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 8: 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. Odgovor na to vprašanje podajamo v nadaljevanju. S pomočjo dodatnih algoritmov, se izkaže kot najboljše iskanje z minimalnim iskalnim oknom (okno z najmanjšo možno velikostjo).

4.6 Iskanje na osnovi glavne variante

Iskanje na osnovi glavne variante (Principal Variation Search) predstavlja malo izboljšavo alfa-beta algoritma. Pri preiskovanju v kontekstu alfa-beta algoritma razlikujemo naslednja vozlišča [10]:

- alfa vozlišča,
  Vozlišča, ki imajo manjšo ali enako vrednost ocenitve glede na alfa mejo. Ta vozlišča so v trenutnem vozlišču za igralca na potezi slabša kot trenutno najboljše izbrano vozlišče.

- beta vozlišča in
  Vozlišča, ki imajo večjo vrednost ocenitve glede na beta mejo. To so vozlišča, ki imajo oceno slabšo od trenutno najboljšega izbranega vozlišča v očetovskem vozlišču. Tako ta vozlišča klestijo iskalno drevo oz. povzročajo rez.

- vozlišča glavne variante.
  Vozlišča, ki imajo vrednost ocenitve znotraj intervala iskalnega okna in postanejo del glavne variante.

Prvi tip vozlišč je neugoden za igralca na potezi, drugi tip vozlišč pa za igralca, ki ni na potezi. Tako ta vozlišča ne vplivajo na končni rezultat iskalnega algoritma. Tretji tip vozlišč pa predstavljajo vozlišča glavne variante in izboljšujejo oceno igralca, ki je na potezi.

Algoritem iskanja na osnovi glavne variante temelji na tem, da najprej poišče glavno varianto. Nato pa preiskuje z minimalnim oknom. Minimalno okno (minimal window) je okno $alfa$, $alfa + \epsilon$, kjer ϵ predstavlja najmanjšo možno razliko med dvema ocenitvama. Npr., če ocenitvena funkcija podaja oceno v obliki celih števil, tedaj bo ϵ imel vrednost 1. Z uporabo iskanja iskanja z minimalnim oknom, algoritemnposkuša izboljšati izbrano glavno varianto [4]. V primeru, ko je rezultat iskanja z minimalnim oknom izven iskalnega okna, potem to vozlišče predstavlja alfa ali beta vozlišče in ne

izboljšuje glavne variante. Če pa se rezultat nahaja znotraj iskalnega okna, obstaja možnost, da preiskano vozlišče postane del glavne variante. Zato je iskanje potrebno ponoviti z večjim oknom. V primeru, da je ocenitev tudi tega iskanja znotraj iskalnega okna, vozlišče postane del glavne variante.

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

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

    // Piščemo glavno varianto
    p.makeNextMove(moves);
    best = -negaScout(p,depth-1,-beta,-alpha);
    p.unmakeMove();
    if( best >= beta ) return best;
    lower = Math.max(alpha,best);

    // Poiščemo najboljšo potezo
    while( p.makeNextMove(moves) ){
        // Iskanje z minimalnim oknom
        value = -negaScout(p,depth-1,-lower -1,-lower);

        // Ali je vozlišče kandidat glavne variante
        if( value > lower && value < beta )
            value = -negaScout(p,depth-1,-beta,-value); // Ponovimo iskanje
        p.unmakeMove();

        // Ali vozlišče pripada glavni varianti
        if( value > best ){
            if( value >= beta ) return value;
            best = value;
            lower = Math.max(alpha,best);
        }
    }
    return best;
}
```

Algoritem 9: NegaScout

Primer iskanja na osnovi glavne variante predstavlja algoritem "NegaScout" (algoritem 9). Ta algoritem pohitri preiskovanje glede na algoritem alfa-beta približno za 10%

[10]. Razlog temu je preiskovanje z minimalnim oknom, ki v preiskovanju povzroča znatno klestenje iskalnega drevesa. Ponovnih preiskovanj pa imamo relativno malo, saj se iskalano okno ob vsaki izboljšavi glavne variante skrči.

4.7 Algoritem MTD(f)

MTD(f) (Memory-enhanced Test Driver with value f) je novejši algoritem, ki pravtako temelji na alfa-beta algoritmu [4, 3]. Prikazuje ga algoritem 10. Algoritem je zelo enostaven in temelji na iskanju z minimalnim oknom in transpozicijsko tabelo. S pomočjo tega iskanja se v MTD(f) algoritmu znotraj zanke določata spodnja in zgornja meja oz. zmanjšuje se interval preiskovanja. Kadar se velikost intervala zmanjša na nič (vrednosti mej se izenačita), pomeni, da smo našli rezultat. Vhodni argument algoritma f pa predstavlja inicijalizacijsko vrednost iskalnih mej. Ponavadi je ta vrednost pridobljena iz prejšnje iteracije iskanja. Tako ima algoritem zelo dobro začetno vrednost, na osnovi katere s pomočjo nekaj iskanj, doseže pravilen rezultat za trenutno globino iskanja.

```
int MTD(Position p, int depth, int f){
    int bound, value = f;
    int alpha = -Evaluate.INFINITY;
    int beta = Evaluate.INFINITY;
    do {
        if( value == alpha ) bound = value + 1;
        else bound = value;
        value = minimalWindowSearch(p,depth,bound);
        if( value < bound ) beta =value;
        else alpha = value;
    }while( alpha != beta ) ;
    return value;
}
```

Algoritem 10: MTD(f)

Algoritem iskanja z minimalnim oknom prikazuje algoritem 11. Ta algoritem je zelo podoben alfa-beta algoritmu, razlika je le v velikosti iskalnega okna. Tako ta algoritem na vhodu sprejme le en parameter. Ta parameter je lahko npr. beta vrednost. Ker pa preiskujemo s pomočjo minimalnega okna, je vrednost alfa enaka beta-1.

```
int minimalWindowSearch(Position p, int depth, int beta){
    int value, best = -Evaluate.INFINITY;
    // 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 = -minimalWindowSearch(p,depth-1,-beta+1);
        p.unmakeMove();
        if( value > best ){
            if( value >= beta ) return value;
            best = value;
        }
    }
    return best;
}
```

Algoritem 11: Iskanje z minimalnim oknom

Kot algoritem iskanja z minimalnim oknom, lahko uporabimo tudi katerikoli drugi alfa-beta temelječ algoritm. Uporabljati ga moramo tako, da mu nastavimo velikost okna na minimalno okno. Tako lahko že obstoječe algoritme na hiter in enostaven način uporabimo znotraj algoritma MTD(f).

Algoritem MTD(f) je bolj zmogljiv v primerjavi z katerm koli drugim algoritmom, ki uporablja iskanje z minimalnim oknom. Z uporabo transpozicijske tabele, pa se zmogljivost algoritma še poveča. Čeprav algoritem dokazuje sam sebe, ga v praksi zelo redko srečujemo. Razlog temu je problem pri implementaciji, kako zbrati glavno varianto in kako obravnavati dvoumne meje, ki so posledica nenatančnih razširitev iskanj.

4.8 Zbiranje glavne variante

Poleg ocene pozicije, iskalni algoritem mora podati tudi izbrano potezo oz. seznam potez (glavno varianto), ki jih je izbral. Da bi dobili glavno varianto lahko uporabimo

algoritme zbiranja glavne variante. Primer zbiranje glavne variante v kontekstu alfa-beta algoritma, prikazuje algoritem 12. Algoritem deluje po principu lepljenja glavnih variant. Lepljenje se izvaja med glavno varianto, ki jo dobimo v obliki argumenta in podvariantami glavne variante, ki jih dobimo s preiskovanjem otrok trenutnega vozlišča.

```
int alphaBeta(Position p, int depth, int alpha, int beta, int[] pv){
    int lastMove, value;
    int [] subPV = new int[depth];
    // Pogoj za končanje rekurzije
    if(  p.endGame() || depth <= 0 ){
        pv[0] = 0;                          // Določimo konec glavne variante
        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,subPV); // Izračun podvariant
        lastMove = p.unmakeMove();
        if( value > alpha ){
            if( value >= beta ) return value;
            // Zbiranje glavne variante
            pv[0] = lastMove;
            System.arraycopy(subPV,0,pv,1,subPV.length-1);
            alpha = value;
        }
    }
    return alpha;
}
```

Algoritem 12: Zbiranje glavne variante

Glavno varianto zbiramo na naslednji način. Kadar smo v listih iskalnega drevesa, glavno varianto inicializiramo. V primeru alfa in beta vozlišč lepljenja ni, saj ta vozlišča ne predstavljajo del glavne variante. Torej lepljenje izvajamo le v primeru vozlišč glavne variante. Ta algoritem je relativno hiter, zahteva pa nekoliko več pomnilnika. Razlog temu je zbiranje glavne variante, ki uporablja sklad rekurzivnih klicev iskalnega algoritma.

Nekoliko boljša rešitev je uporaba transpozicijske tabele. Ta tabela vsebuje ključe in najboljše poteze preiskanih pozicij. Tako s pomočjo njene vsebine lahko zberemo glavno varianto. Ta način zbiranja glavne variante je hitrejši od prejšnjega, zahteva pa več pomnilnika in določeno implementacijo transpozicijske tabele.

4.9 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. 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 13.
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. 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
    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 13: Iskanje mirovanja

4.10 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, tako 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, ubijalska hevristika in hevristika zgodovine.

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 [10, 6]:

- ključ,
  Ključ predstavlja ključ določene pozicije. Te ključe smo predstavili v podpoglavju 3.8 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 spodnjo mejo 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 [6].

- 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. Pridobljena ocena 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 spodnja meja ima 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 14.

Računalniki imajo omejen pomnilnik, tako je tudi velikost transpozicijske tabele omejena. To pa pomeni, da v določenih primerih prihaja do kolizij. Problem kolizij rešujemo s pomočjo shem zamenjav. Le te pa temeljijo na naslednjih konceptih [6]:

- Koncept *globine* (Concept *Deep*)
  To je tradicionalen koncept in temelji na globini preiskanega poddrevesa. V primeru kolizije, ostane v tabeli pozicija z večjo globino.

- Koncept *novega* (Concept *New*)
  Koncepth novega daje prednost novim pozicijam. V primeru kolizije, se podatki stare pozicije nadomestijo s podatki nove pozicije.

- Koncept *starega* (Concept *Old*)
  Koncept daje prednost starejšim pozicijam. V primeru kolizije se podatki ne prepišejo.

- Koncept *velikega* (Concept *Big*)
  Koncept velikega temelji na številu preiskanih vozlišč v poddrevesu. V primeru kolizije, ostanejo v tabeli podatki pozicije, ki vsebuje več preiskanih vozlišč v poddrevesu.

- Koncept *dveh nivojev* (Concept *Two-level*)
  Koncept dveh nivojev temelji na dvonivojski trenspozicijski tabeli. V primeru kolizije imamo dve možnosti:

  – *v primeru kadar ima nova pozicija večjo globino poddrevesa, se vrednosti iz tabele prvega nivoja prepišejo v tabelo drugega nivoja. Vrednosti nove pozicije pa se zapišejo v tabelo prvega nivoja;*

  – *v nasprotnem primeru se podatki pozicije shranijo v tabelo drugega nivoja.*

Katerega od konceptov bomo uporabili je odvisno od razpoložljivega pomnilnika. V primeru kadar nimamo dovolj pomnilnika, je najbolje uporabiti koncept dveh nivojev. Najbolj uporabljen koncept pa je koncept globine. Koncept starega pa se ne uporablja pri šahovskih igrah.

4.11 Zgodovinska hevristika

Glede na že preiskane pozicije, si pri generiranju zaporedja potez lahko pomagamo z ubijalsko in zgodovinsko hevristiko. Pri preiskovanju se zelo pogosto zgodi, da poteza, ki povzroči rez v določenem vozlišču, povzročajo rez tudi v sosednjih vozliščih. Tako si lahko te poteze shranimo in v sosednjih vozliščih preiskovanje začnemo s pomočjo teh potez. Opisan način oblikovanja zaporedja potez imenujemo ubijalska hevristika. Razširitev te hevristike predstavlja zgodovinska hevristika. Dobra poteza nekega vozlišča, predstavlja tudi dobro potezo v vozliščih, ki jih bomo preiskali v nadaljevanju. Npr., v nekem vozlišču poznamo najboljšo potezo. Potem obstaja velika verjetnost, da bomo v iskalnem drevesu najdli vozliče, ki se od prejšnjega vozlišča s stališča pozicije, razlikuje le v položaju ene figure. Ta mala razlika ponavadi bistveno ne spremeni pozicije in s pomočjo prej izbrane poteze, lahko oblikujemo zelo dobro zaporedje potez.

```
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 z 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 14: Alfa–beta z transpozicijsko tabelo

V okviru alfa-beta algoritma lahko za kasnejšo uporabo shranjujemo dva tipa potez. To so poteze, ki predstavljajo najbolšo izbrano potezo in poteze, ki povzročajo rez. Tako shranjene poteze niso nujno tudi najboljše. Zato si lahko vodimo zgodovinsko uspešnost (zgodovinsko hevristiko) in na osnovi nje urejamo nova zaporedja potez. Dodatno pa moramo upoštevati še globino iskanja. Kajti poteze izbrane blizu korenskega vozlišča so pomembnejše od tisth, ki so izbrane blizu listov.

4.12 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 z 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 pozicij, kadar odigramo ničelno potezo, dobimo neveljavno pozicijo. Zato je iskanje mirovanja potrebno omejiti, le na pozicijie, ki niso v šahu. Dodaten problem predstavljajo še primeri, kadar se zaporedoma odigrajo samo ničelne poteze. Tako dobimo degradirano iskanje. Rešitev tega problema pa predstavlja, 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 dodtnimi 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.

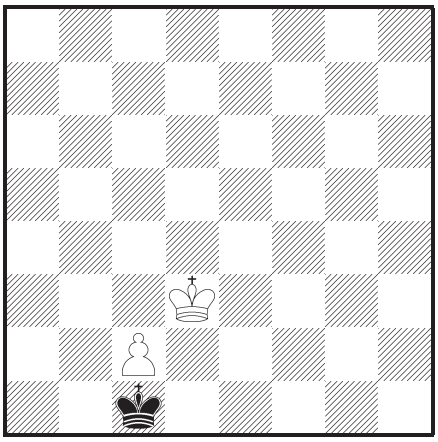
```
int R = 2;
int alphaBeta(Position p, int depth, int alpha, int beta){
    int value;
    // Pogoj za končanje rekurzije
    if( p.endGame() || depth <= 0 ) return p.evaluate();

    // 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 15: Klestenje z ničelno potezo

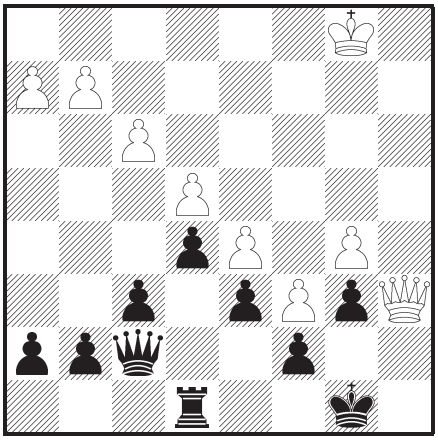
Algoritem iskanja z ničelno potezo prikazuje algoritem 15. Algoritem omejuje iskanje mirovanja na pozicije, ki niso v šahu. Poleg reduciranja faktorja globine nad iskanjem z ničelno potezo, uporablja še iskanje z minimalnim oknom. Dobljen rezultat iskanja pa nato preveri, ali je lahko del glavne variante. 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 ničelene poteze pa 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 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 črneg z potezo Qg2. Pri preiskovanju z globino iskanja 2, se z ničelno potezo spremeni igralec na potezi in zmanjša 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 ničelene 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 spregledea, lahko na različne načine reduciramo. Prvi način je uporaba

adaptivnega faktorja reduciranja. To pomeni, da faktor reduciranja prilagajamo tipu pozicije in globini iskanja. Dru način pa predstavlja verifikacijo klestenja z ničelno potezo. Ta algoritem podrobneje predstavljamo v naslednjem podpoglavju.

4.13 Verifikacija klestenja z ničelno potezo

Izboljšavo klesanja z ničelno poteza predstavlja algoritem verifikacije klestenja z ničelno potezo. Ta algoritem poleg že omenjenih omejitev, ki smo jih spoznali pri klestenju z ničelno potezo, vpeljuje verifikacijo. Tako na osnovi rezultata ničelne poteze, algoritem dodatno preverja pravilnost rezultata. Opisani algoritem prikazuje algoritrem 16.

Ta algoritem v mnogih primerih odkrije napačno ocenjene pozicije, ki jih povzroči ničelna poteza. V primeru, kadar odkrijemo napačno ocenjeno pozicijo, iskanje ponovimo brez ničelne poteze in izognemo se napačnim ocenam. Ta algoritem je relativno preprosto implementirati. Prav tako program, ki ima implementiran algoritem iskanje z ničelno potezo, lahko z dodajanjem nekaj vrstic, dopolnimo do predstavljenega algoritma z verifikacijo.

Predstavljena algoritma, ki klestita iskalno drevo s pomočjo ničelne poteze, preiskujeta manjše iskalno drevo in v enakem času lahko preiščeta drevo z večjo globino. Obstaja pa nevarnost, da spregledata pomembne pozicije in tako izbereta napačno potezo. Število takih pozicij lahko zmanjšali s pomočjo adaptivnega faktorja reduciranja in verifikacije. Ti dve omejitvi nam dodatno omogočata uporabo večjega faktorja reduciranja. Posledica tega pa je manjše iskalno drevo in hitrejši iskalni algoritem.

4.14 Dodatne izboljšave

Zmogljivost predstavljenih algoritme lahko izboljšamo na različne načine. Npr. v korenskem vozlišču iskalnega drevesa si za vsako preiskano potezo, lohko shranjujemo glavno varianto (Refutation tables). Določeno glavno varianto pa lahko nato uporabimo v naslednji iteraciji. Uporabimo jo tako, da najprej poskušamo preiskovati z njenimi potezami.

```
int R = 3;
int alphaBeta(Position p, int depth, int alpha, int beta, boolean verify){
    int value;
    boolean fail = false ;

    // Pogoj za končanje rekurzije
    if( p.endGame() || depth <= 0 ) return p.evaluate();

    // Iskanje z ničelno potezo in minimalnim oknom in faktorjem reducirnja R
    if( !p.inCheck() && p.nullMoveOk() && (!verifay || depth > 1) ){
        p.makeNullMove();
        value = -alphaBeta(p,depth-1-R,-beta,-beta+1,verify);
        p.unmakeNullMove();
        if( value >= beta ){
            if( verify ){
                depth - -; // Reduciranje globine iskanja
                verify = false ; // Verifikacija v poddrevesih ni potrebna
                fail = true ; // Za odkrivanje napačno ocenjenih pozicij
            }
            else return value;
        }
    }

    while( true  ){
        // Del alfa-beta algoritma
        while( p.makeNextMove(moves) ){
            value = -alphaBeta(p,depth-1,-beta,-alpha, verify);
            p.unmakeMove();
            if( value > alpha ){
                if( value >= beta ) return value;
                alpha = value;
            }
        }
        // Kadar je samo ničelna poteza povzročila rez
        if( fail && alpha < beta ){
            depth ++; // Ponovimo iskanje z prvotno globino
            fail = false ; // Ponovna verifikacija v tem vozlišču ni potrebna
            verify = true ; // Verifikacija v poddrevesih je potrbna
            continue ;
        }
        break ;
    }
    return alpha;
}
```

Algoritem 16: Verifikacija klestenje z ničelno potezo

V primeru uporabe algoritmov z transpozicijsko tabelo in določenih drugih primerih, lahko zmogljivost algoritmov izboljšamo tako, da preiskujemo tudi v času kadar nismo na potezi. Tako si algoritem lahko pripravil podatke oz. vsebino tabele, ki jo lahko zelo učinkovito uporabi v naslednji potezi oz. kadar bo izbiral svojo najboljšo potezo.

Dodatno lahko zmogljivost algoritmov povečamo še z uporabo porazdeljenega računalniškega sistema. Breme algoritma lahko porazdelimo na več računalnikov. Tako algoritem ima na voljo več računalniških virov s pomočjo katerih lahko dosega boljše rezultate.

Zmogljivost iskalnih algoritmov je odvisna tudi od ocenitvene funkcije. Tako s pomočjo izboljšave ocenitvene funkcije, lahko izboljšamo tudi zmogljivost iskalnih algoritmov. Podrobnejši opis ocenitvene funkcije in njeneih lastnosti predstavljamo v poglavju 6.

Za doseganje čim večjih iskalnih globin lahko uporabimo različne algoritme, ki jih je potrebno združiti v eno celoto. Določeni algoritemi se izkažejo kot boljši v določenih fazah igre, določenih globinah preiskovanja itd. Zato jih je potrebno združiti v eno celoto, ki nam omogoča učinkovito preiskovanje iskalnega drevesa. Npr. v eno celoto lahko združimo naslednje algoritme:

- aspiracijsko iskanje,
  Omogoča časovno omejeno iskanje in iterativno poglabljanje.
- MTD(f),
  Omogoča upravljenja alfa beta iskanja z ničelnim oknom na osnovi aspiracijske vrednosti, pridobljene iz prejšnje iteracije aspiracijskega iskanja.
- alfabeta algoritem z ničelnim oknom in transpozicijsko tabelo in
  Centralni algoritem iskanja, ki temelji na alfa-beta algoritmu in uporablja ničelno okno ter transpozicijsko tabelo.
- iskanje mirovanja.
  Uporablja ga alfa-beta algoritem za ocenjevanje dinamičnih pozicij.

Tako vidimo, da je zmogljivost šahovski algoritmov odvisna tudi od izbire in načina združevanja podalgoritmov, ki pa so spet odvisni od zelo dosti dejavnjikov. Ne smemo pa pozabiti, da na zmogljivost vpliva še ocenitvene funkcije, ki predstavlja osnovo iskalnih algoritmov šahovskih programov.

5. Generator potez

Generator potez je ena od zelo pomemebnih komponent računalniškega šaha. Implementiramo ga na osnovi predstavitve igre. Njegova naloga je generiranje "dobrega" zaporedja potez, v čim krajšem času. Dobro zaporedje potez omogočilo iskalnemu algoritmu, oblikovanje čim manjšega iskalnega drevesa. Tako v primeru idealnega generatorja potez lahko dobimo minimalno iskalno drevo. V realnosti tega ne moremo doseči. Z uporabo določenih algoritmov in mehanizmov generiranja potez, lahko oblikujemo iskalno drevo, ki je od 20% - 30% večje od minimalnega iskalnega drevesa.

Kako implementirati generator potez? Izbiramo lahko med dvema načinoma. Prvi način predstavlja generator, ki sproti generira poteze. Tako v primeru reza v iskalnih algoritmih, preostale poteze ni potrebno generirati. Drugi način pa, predstavlja generator, ki vse poteze generira na enkrat. V tem primeru imamo vse poteze na voljo. Dobljene poteze nato na določen način lahko uredimo, ter dobimo "dobro" zaporedje potez. Drugi način se izkaže kot boljši. Le ta s pomočjo dobrega zaporedja potez, omogoča iskalnim algoritmom, dosegajo znatno boljših rezultatov.

Pri implementaciji generatorja, največji problem predstavlja odkrivanje, ali je generirana poteza veljavna. Ta problem še posebej pride do izraza, kadar moramo preverjati ali generirana poteza povzročila šah igralcu, ki odigra potezo. Razlog temu je komleksno in časovno zahtevno ugotavljanje ali je igralec v šahu. Ta problem lahko rešimo na dva načina. Prvi način v času generiranja potez, odigra generirano potezo in preveri ali je poteza legalna. Drugi način pa generira vse poteze, tako veljave kot neveljavne. Veljavnost potez pa se preveri, v času izvajanja iskalnega algoritma. Tako se časovno zahtevno preverjanje, ali je igralec v šahu, izvaja v času preiskovanja iskalnega drevesa. Zaradi rezov v iskalnih algoritmih, določene poteze ne bomo preiskovali s tem pa tudi ne bomo preverjali ali so veljavne. Na tak način, dobimo generator potez, ki je časovno manj potraten.

Poglejmo še kako oblikovati dobro zaporedje potez? Najbolj preprost in zmogljiv mehanizem za oblikovanje dobrega zaporedja potez, predstavlja MVV/LVA (Most Valuable Victim / Least Valuable Attacker) shema. Ideja te sheme, je da uredi poteze po naslednjem kriteriju. Na začetku seznama so poteze, ki vsebujejo jemano figuro z največjo vrednostjo in figuro poteze z najmanjšo vrednostjo. S pomočjo te sheme se zmogljivost iskalnih algoritmov znatno poveča. Dodatno pri oblikovanju zaporedja potez lahko uporabimo še transpozicijsko tabelo ter ubijalsko in zgodovinsko hevristiko. Tako lahko na osnovi že preiskanih pozicij, in shranjenih podadatkov v pomnilniku, dodatno izboljšamo zaporedje potez.

Vse naštete mehanize za oblikovanje dobrega zaporedja potez, je na kocu potrebno še združiti v eno celoto. Najbolj uporabljen način predstavlja, uporabo metod po naslednjih prioritetah:

1. poteza transpozicijske tabele,
2. poteze jemanja v kontekstu MVV/LVA zaporedja,
3. poteze ubijalske hevristike,
4. poteze zgodovinske hevristike in
5. poteze na osnovi statičnih hevristik.

V tej shemi prvo potezo predstavlja poteza transpozicijske tabele. To je poteza, ki smo jo za to pozicijo že preiskali in izbrali kot najboljšo ali kot potezo, ki je povzročila rez. Poteze, ki sledijo tej potezi, so poteze jemanja oz. poteze urejene po prej omenjene sheme MVV/LVA. Tem potezam sledijo poteze ubijalsek hevristike. To so poteze, ki so v prejšnjih vozliščih povzročale rez. Tako obstaja določena verjetnost, da bodo ubijalske poteze tudi v tej poziciji povzročile rez. Naslednje poteze pa predstavljajo poteze, ki jih uredimo po zgodovinski hevristiki oz. njihovi uspešnosti v že preiskanih pozicijah. Preostale poteze pa lahko uredimo na osnovi statičnih hevristik. Npr. poteze, ki premikajo figure na sredino deske so ponavadi boljša od tistih, ki figuro premikajo v kot.

Kot vidimo je generator potez zelo pomembna komponenta računalniškega šaha. Z dodatnimo mehanizmi pa lahko zaporedje potez še dodatno izboljšamo. Posledica tega

je nekoliko počasnejši generator potez ter iskalni algoritmi, ki dosegajo večje iskalne globine. Tako vidimo, da dobro zaporedje potez oz. generator potez znatno vpliva na zmogljivost šahovskih programov.

6. Ocenitvena funkcija

Ocenitvena funkcija (Evaluation Function) podaja statično oceno pozicije 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.

6.1 Izrazi ocenitvene funkcije

Ocenitveno funkcijo sestavlja več izrazov. Vsak izraz pa ocenjuje določene specifičnosti pozicije. Tako lahko v ocenitveni funkciji najdemo izraze za določanje naslednjih verjetnosti: o zmagi, o zmagi v nekaj potezah in o zmagi v končnici igre. Če verjetnost za zmago igralca z belimi figurami označimo z ws in za zmago njegovega nasprotnika

bs, verjetnost za zmago v nekaj potezah z wm in bm, in verjetnost za zmago v končnici igre z we in be, potem je verjetnost za zmago v določeni poziciji za igralca z belimi figurami enaka:

$$ws + (1 - bs - ws) \bullet wm + (1 - bs - ws - bm - wm) \bullet we \tag{6.1}$$

oz. za njegovega nasprotnika:

$$bs + (1 - bs - ws) \bullet bm + (1 - bs - ws - bm - wm) \bullet be \tag{6.2}$$

Tako v ocenitveni funkciji poleg predstavljenega primera, lahko uporabljamo še druge verjetnostne izraze in jih med seboj kombiniramo. Tukaj se postavi vprašanje, katera kombinacija je najboljša? To ugotovljamo tako, da program testiramo z različnimi kombinacijami izrazov.

6.2 Informacije v ocenitveni funkciji

Kot smo že povedali, ocenitveno funkcijo sestavlja več izrazov. Ti izrazi pa lahko odkrivajo naslednja znanja:

- material,
  Razlika vsote vrednosti belih figur in vsote vrednosti črnih figur. Za igralca je boljše, če je vsota vrednosti njegovih figur večja od nasprotnikovih.

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

- okupacija,
  Polja v igri so lahko nadzorovana s strani belih, črnih, belih in črnih, in nobenih figur. Tako lahko polja razdelimo na tista, ki so okupirana s strani belih figur, črnih figur in neutralna polja. Za igralca je bolje, če njegove fugure okupirajo večj polj, kot nasprotnikove.

- grožnja
  Predstavlja možnost, da eden od igralcev na nek način, grozi z določeno potezo drugemu igralcu. Npr., s premikom določene figure, lahko nasprotnik doseže šah, ki pa mu nato omogoča nadaljnji razvoj igre in napad.

- struktura in
  Določena struktura figur je boljša v primerjavi z drugo. Npr., dva kmeta drug za drugim predstavljata slabo strukturo. V kasnejši fazi igre ali končnici lahko kmeta postaneta, zelo lahek plen za nasprotnika. Iskalni algoritem je omejen z globino iskanja in ne more doseči končnice igre. Tako se lahko zgodi, če ocenitvena funkcija ne vsebuje znanja o sturkturi, da program odigral slabo potezo.

- vzorci.
  Določeno znanje lahko vgradimo s pomočjo vzorcev in tako dosežemo boljšo ocenitev pozicije. Npr. s pomočjo vzorca, lahko ugotovimo kakšna je zaščita kralja.

6.3 Uglaševanje ocenitvene funkcije

Ocenitvena funkcija glede na določene vrednosti izrazov in spremenljivk podaja ocenitev pozicij. Tukaj se postavlja naslednje vprašanje: kako pridemo do vrednosti spremenljivk in izrazov? V ta namen se lahko uporabijo naslednje metode:

- normalizacij,
  Glede na določeno vrednost, npr. material kmeta, lahko izračunamo vrednost materiala vseh kmetov. To naredimo s pomočjo množenja materiala enga kmeta z številom kmetov na deski. Tako je rezultat izraza odvisen od samo enega parametra.

- izvajanje pritiska in
  Z izbiro določenih vrednosti parametrov, lahko v različnih pozicijah dosežemo določen cilj. Npr, zamenjava trdnjave za konja ali lovca je slaba, celo v primeru kadar pridobimo še enega kmeta. Toda v primeru kadar pridobimo dva kmeta se ta zamenjava izkaže kot dobra. Tako vrednosti figur morajo zadostiti naslednjima izrazoma: $R > B + P$ *in* $R < B + 2 \bullet P$. *Kjer* R *predstavlja vrednost trdnjavo,* B *vrednost lovca in* P *vrednost kmeta.*

- upravljanje prevar.
  V določenih pozicijah, lahko poskušamo z spreminjanjem paramterov ocenitvene funkcije, ugotoviti kakšna je pozicija. Tako lahko v primeru silovitega napada, poskušamo z žrtvovanjem figur doseči mat.

Uglaševanje ocenitvene funkcije poleg že opisanih metod, lahko izvajamo tudi brez človeškega posredovanja. To naredimo tako, da uporabimo naslednje metode strojnega učenja:

- vzpenjene na hrib *(Hill-climbing)*,
  Ta metoda, periodično spremenija prarametre za neko malo vrednost in testira zmogljivost spremenjenih parametrov. Spremenjene parametre obdrži le v primeru kadar se zmogljivost programa poveča. Slabost te metode je njena počasnost in možnost, da optiči v lokalnem optimumu.

- simulirano ohlajanje *(Simulated annealing)*,
  Ta metoda je podobna vzpenjanju na hrib. S pomočjo malih sprememb v parametrih in ocenjevanju le teh, obdržimo le tiste parametre, ki izboljšajo zmogljivost programa. Toda če se zmogljivost ne izboljšuje, se naključno ali po določenem pravilu lahko, izberejo tudi slabši parametri. Tako se izognemo lokalnim optimumom. Verjetnost izbiranja slabših parametrov je na začetku velika, nato pa se postopoma zmanjšuje. Tako na nek način simuliramo ohlajanje, po čem je metoda tudi dobila ime. Ta metoda je počasnejša od metode vzpenjanja na hrib omogoča pa pridobivanje boljših rezultatov

- genetski algoritmi *(Genetic algorithms)* in
  Vzpenjanje na hrib in simultano ohlajanje obdržita samo eno dobro množico parametrov, ki se je postopoma spreminjala. Genetski algoritmi pa vsebujejo populacijo množic. Z operatorji mutacijo, križanjem in selekcijo, populacijo na določen način spreminjamo ter dobijamo nove množice parametrov [11]. Te množice parametrov nato ocenimo ter jih glede na ocenitev zavržemo ali dodamo v populacijo. Ta proces spreminjanja nato periodično ponavljamo, do določenega končnega pogoja. Tako na koncu dobimo določeno populacijo "dobrih" parametrov ocenitvene funkcije.

- nevronske mreže *(Neural networks)*.
  S pomočjo določene nevronske mreže lahko nadomestimo celotno ocenitveno funkcijo. Nevronska mreža vsebuje nevrone in uteži na osnovi katerih podaja oceno pozicije. Kako pa nastaviti uteži nevronski mreži. To lahko naredimo s pomočjo že omenjenih metod strojnega učenja. Tako lahko brez kakršnega koli šahovskega znanja naučimo nevronsko mrežo ocenjevati pozicije. Za to bomo potrebovali nekaj

tednov, toda dobili bomo dobro ocenitveno funkcijo in nebo sa nam potebno ubadati z uglaševanjem ocenitvene funkcije.

Vse predstavljene metode zahtevajo avtomatsko ocenitev programa. To ocenitev programa lahko naredimo na več načinov.

- Program lahko zaženemo nad množico testnih pozicij in primerjamo rešitve pozicij z potezami, izbranimi s pomočjo programa. Tako ocenitev v tem primeru predstavlja število pravilno rešenih pozicij.
- Program lahko igra nasproti drugemu šahovskemu programu, samemu sebi, ali pa proti svojim prejšnjim verzijam. Ocenitev pa predstavlja število zmag in remijev ocenjevanega programa.
- Ocenitveno funkcijo lahko primerjamo z preiskovalnim algoritmom. Pozicijo ocenimo s pomočjo ocenitvene funkcije in s pomočjo iskalnega algoritma. Ocenitev pa predstavlja število enako ocenjenih pozicij.

Opisane metode nam omogočajo na različne načine uglaševanje in implementacijo ocenitvene funkcije. Kot zelo zanimiva metoda se izkaže uporaba genetskih algoritmov za uglaševanje ocenitvene funkcije v obliki nevronske mreže. Ta metoda omogoča samodejno učenje, ki pa traja zelo dolgo. Te lastnosti pa vsebujejo tudi vse preostale metode strojnega učenja.

7. Implementacija

7.1 Načrtovanje

Predstavljene mehanizme in algoritme računalniškega šaha smo implementirali in preiskusili. V ta namen smo načrtovali in implementirali šahovski program. Načrtovali smo ga tako, da omogoča enostavno dodajanje in testiranje iskalnih algoritmov in različnih predstavitev igre. To smo dosegli s pomočjo aplikaciskih vmesnikov in dedovanjem. Tako smo omogočili hiter način dodajanja novih predstavitve igre in iskalnih algoritmov in uporabo že implementiranih ostalih sestavnih delov računalniškega šaha. Npr. če želimo preiskusiti nov iskalni algoritem. Tedaj bomo s pomočjo dedovanja, implementirali določene abstraktne metode določenega abstraktnega razreda. Pri implementaciji tega algoritma pa nam ni potrebno skrbeti o implementaciji predstavitve igre, transpozicijski tabeli, komunikacijskem protokolu, merjenju časa itd. Tako načrtovan program, omogoča tudi implementacijo različnih predstavitev igre neodvisno od iskalnih algoritmov. V primeru, če implementiramo novo predstavitev igre, lahko na osnovi te predstavitve, uporabimo vse obstoječe iskalne algoritme.

Program smo načrtovali tako, da so posamezne komponente računalniškega šaha med seboj neodvisne. Zaradi hitrosti programa in vmesnikov, pa vseh komponent nismo mogli tako načrtovati. Primer takih komponent sta predstavitev potez in figur. Predstavitev potez mora biti kompaktna in jedrnata. Z njimi izvajamo operacije nad pozicijami, z njimi manipuliramo ter jih shranjevati za kasnejšo uporabo. Poteze prav tako predstavljajo vmesnik med iskalnim algoritmom in predstavitvijo igre. Tako vidimo, da predstavitev potez ni možno poljubno spreminjati. Zato smo se določenim, da takim komponentam, določili fiksno obliko.

Program smo načrtovali tako, da ga sestavlja več paketov. Osnovni paket smo poimenovali *chess*. Ta paket vsebuje naslednje pakte: *run, search, uci* in *presentation*. Paket run je namenjen zagonu šahovskega programa. Vsebuje pa razrede, ki omogočajo zagon

šahovskega programa na različne načine (z uporabo različnih predstavitev in algoritmov).

Paket search je namenjen iskalnim algoritmom. Le ta vsebuje abstraktni razred *Search*, ki vsebuje instanco na predstavitev igre, časovnik in komunikacijski protokol ter naslednji abstraktni metodi ***void*** *search()* in ***int*** *getBestMove()*. Tako s pomočjo dedovanja in implementacije tega razreda lahko na hiter način dodajamo in preiskušamo nove iskalne algoritme.

Paket uci predstavlja implementacijo standardnega komunikacijskega protokola UCI. S pomočjo tega protokola, naš progam zna komunicirati z grafičnimi uporabniškimi vmesniki.

Paket presentation pa vsebuje predstavitve igre in transpozicijske tabele. Ta paket vsebuje vmesnik *Position*, abstraktni razred *TranspositionTable* in razreda *Piece* in *Move*. Z implementacijo vmesnika Position lahko implementiramo različne predstavitve pozicije. Z dedovanjem in implementacijo abstraktnega razreda TranspositionTable pa lahko implementiramo različne transpozicijske tabele. Razreda Piece in Move pa vsebujeta metode in konstante za delo z figurami in potezami.

Opisano arhitekturo programa ter implementacijo vmesnikov in abstraktnih razredov našega programa prikazuje razredni diagram na sliki 7.1. S pomočjo te arhitekture lahko preiskušamo različne iskalne algoritme v kombinaciji z različnimi implementacijami predstavitve igre in transpozicijske tabele. Pri tem pa nam ni potrebno skrbeti za njihovo medsebojno povezovanje, komunikacijski protokol in merejenje časa. Zaradi zmogljivosti programa in povezljivosti komponent pa predstavitve določenih komponent (predstavitev potez in figur) niso transparentne.

7.2 Implementacija

Pri implementaciji smo implementirali načrtovano arhitekturo programa. Najprej smo implementirali predstavitev potez in figur. Predstavitev potez smo zasnovali tako, kot smo jih opisali v podpoglavju 3.1. Nato smo implementirali bitno predstavitev igre (razred BitBoard) in transpozicijsko tabelo (razred TTDeep), ki temelji na konceptu

globine. Tako smo implemetnirali paket presentation oz. predstavitev igre.

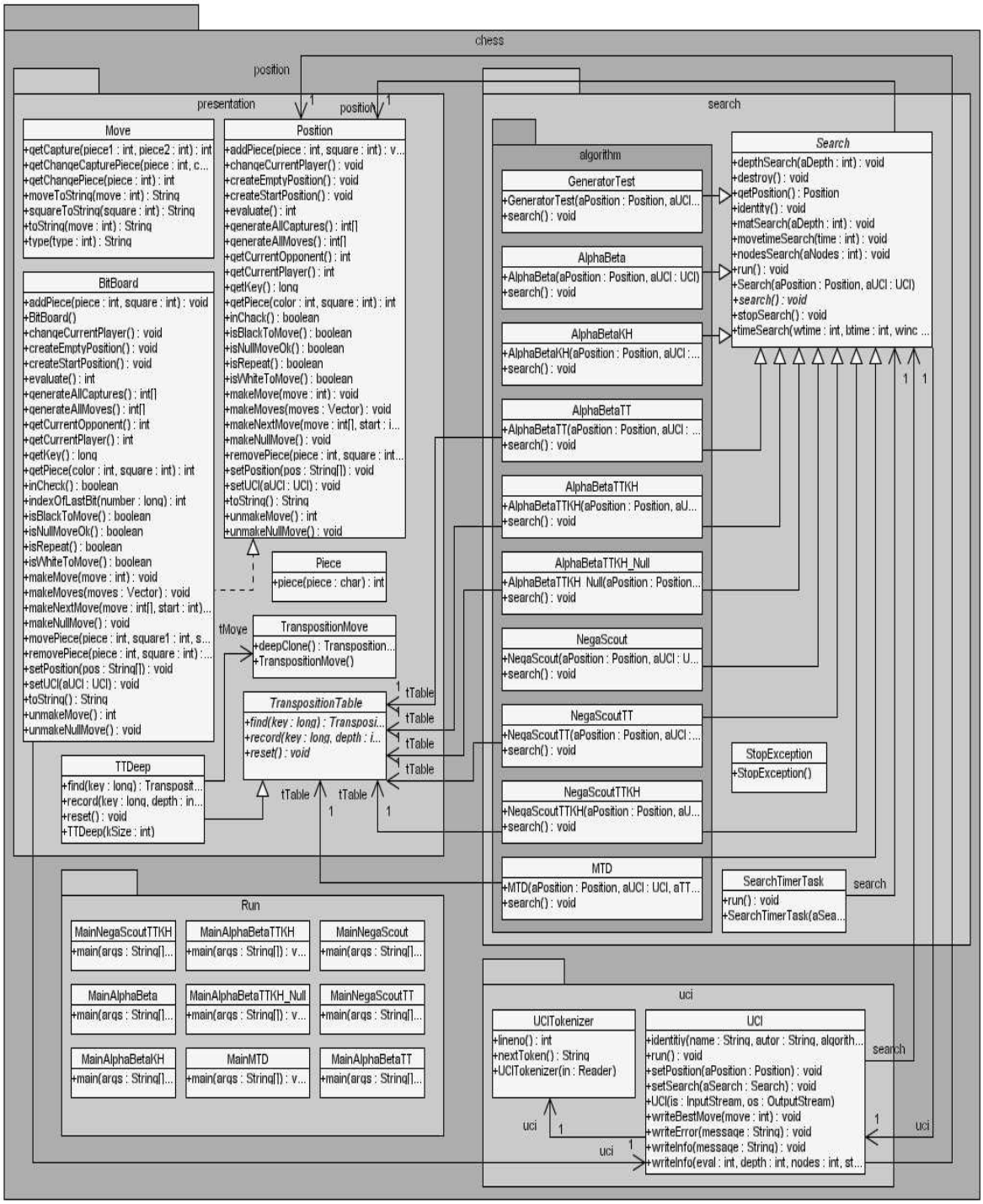
Implementacijo smo nadaljevali z uci paketom. Tukaj smo implementirali UCI protokol (razred UCI) in leksikalni analizator (razred UCITokenizer). Protokol smo implementirali tako, da lahko komunicira z poljubnim vhodnim (InputStream) in izhodnim (OutputStream) tokom.

Pri implementaciji paketa search smo implementirali razreda StopException in SearchTimerTask. Prvi omogiča zaustavljanje niti iskalnega algoritma, drugi pa predstavlja opravilo, ki se zgodi, kadar poteče določen čas. Tako smo s pomočjo teh dveh razredov v razredu Search implementirali časovno omejeno iskanje. Poleg časovno omejenega iskanja, smo v razredu Search implementirali še metode, ki nastavljajo spremenljivke o času iskanja, globini iskanja, številu vozlišč, do katerih naj iskalni algoritem preiskuje itd. Nato smo v podpaketu *algorithm* implementirali z dedovanjem razreda Search, še iskalne algoritme (alfa - beta, NegaScout, MTD(f), itd.).

Na koncu smo implementirali še paket run. Tukoj smo za vsakega od iskalnih algoritmov implementirali razred za zagon programa. Tako smo lahko v grafičnem uporabniškem vmesniku medseboj primerjali različne iskalne algoritme.

Opisano arhitekturo programa v obliki razrednega diagrama prikazuje slika 7.1. V času izvajanja arhitekturo programa sestavljajo tri niti. Prva nit komunicira z grafičnim uporabniškim vemsnikom in koordinira preostali dve niti. Druga nit z pomočjo iskalnih algoritmov preiskuje drevo igre. Tretja nit pa skrbi za časovno omejena iskanja. Tako v primeru, kadar poteče čas, ta nit zauistavi iskalni algoritem in poda izbrano potezo oz. glavno varianto.

Opisano implementacijo smo implementirali v programskem jeziku java. Na tak način smo omogočili, da se naš program lahko izvaja na različnih platformah. Slabost naše implementacije pa je nekoliko počasnjši program. Razlog temu je izbira programskega jezika in objektno orientirano načrtovanje in implementacija. Prednost, ki pa smo jo pridobili je enostavno vzdrževanje in prilagajanje programa novim algoritmom in predstavitvam igre.



Slika 7.1: Razredni diagram

7.3 Grafični uporabniški vmesniki

Pri implementaciji šahovskega programa je zelo pomemben tudi grafični uporabniški vmesnik. Njegova naloga je, da uporabniku omogoči uporabniško prijazno komuniciranje z šahovskim programom. Ker pa se v ozadju izvaja šahovski program, se od grafičnega vmesnika dodatno zahteva, čim manjše obremenjevanje računalniških virov.

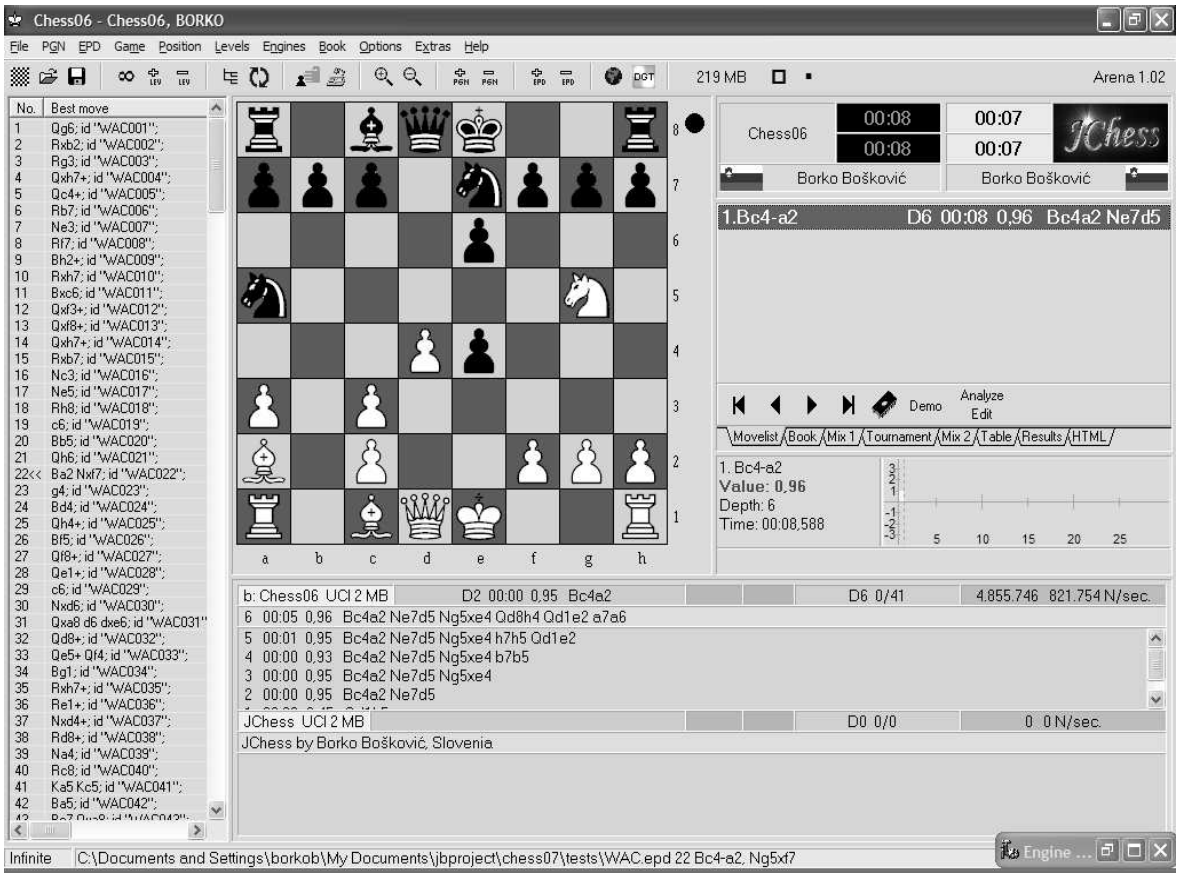
Naš program ne vsebuje grafičnega uporabniškega vmesnika. Ima pa implementiran UCI vmesnik. S pomočjo tega vmesnika lahko naš pogram povežemo z različnimi grafičnimi vmesniki kot so npr. Arena, Jose, Fritz itd.

7.3.1 Arena

Arena je prosto dostopeg grafični uporabniški vmesnik (slika 7.2). Omogoča komunikacijo z šahovskimi programi preko Winboard in UCI protokola. Poleg tega vsebuje še mnogo različnih funkcij, kot so npr. [14]:

- igranje igre proti Winboard ali UCI šahovskemu programu,
- igranje turnirjev med med šahovskimi programi,
- analizo pozicij s pomočjo šahovskih programov,
- podpira Fischer-ov naključni šah,
- vključuje knjižnice za šahovske programe,
- itd.

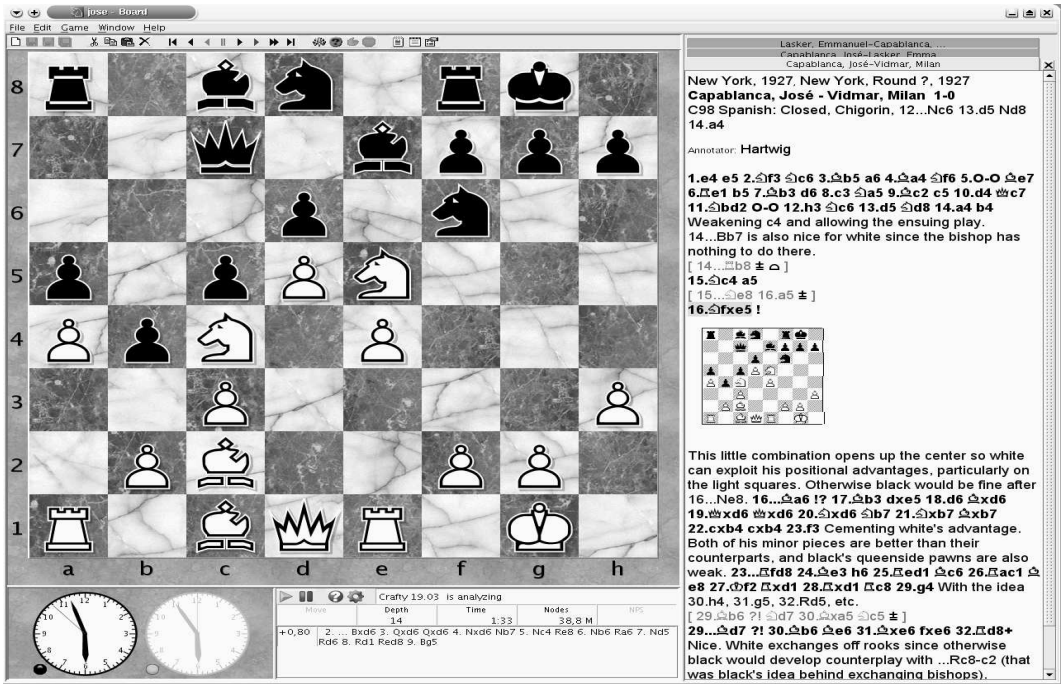
Arena je zelo zmogljiv grafični uporabniški vmesnik, ki vsebuje zelo dosti funkcionalnosti. Napisal ga je Martin Blume v orodju Delphi. Tako ga lahko zaganjamo samo na Windows operacijskih sistemih in vsebuje samo 2D pogled. To pa sta tudi edini slabost, ki jih lahko očitamo temu grafičnemu vmesniku.



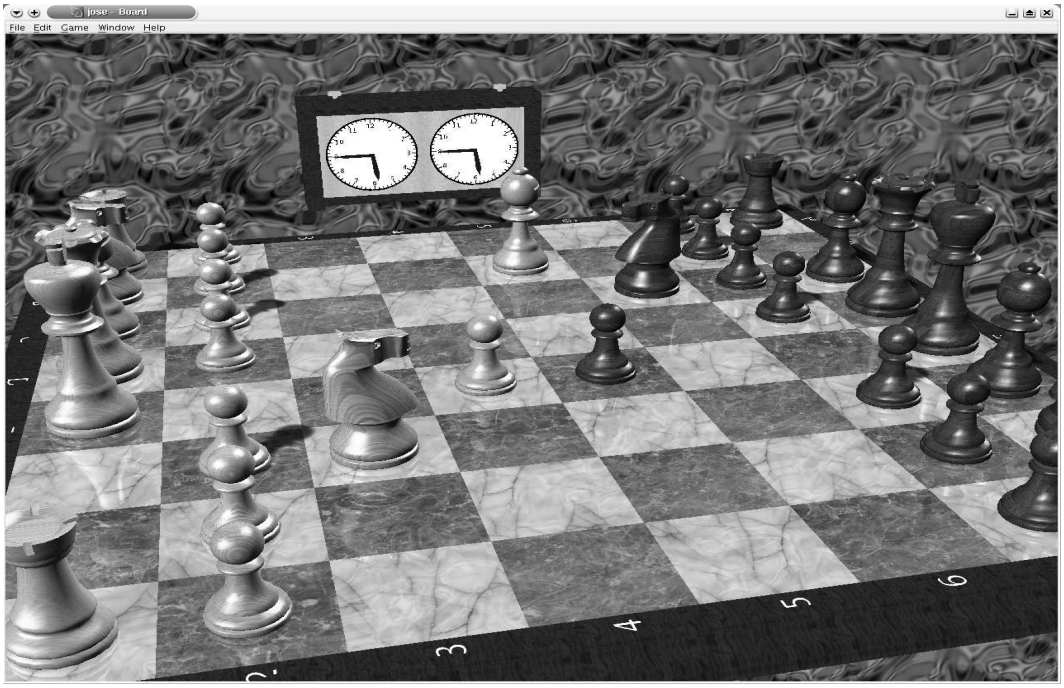
Slika 7.2: Arena

7.3.2 Jose

Jose je grafični uporabniški vmesnik, ki je pravtako prosto dostopen in omogoča igranje igre proti Winboard in UCI šahovskim programom. Ta program je napisan v programskem jeziku java ter uporablja Java3D aplikacijski vmesni in MySql podatkovno bazo [15]. Tako lahko partije shranjujemo v podatkovno bazo in jih kasneje tudi urejamo in analiziramo. Ker pa je napisan v programskem jeziku Java, zahteva nekaj več računalniških virov in je prenosljiv na različne platforme. Od programa Arena se razlikuje še po tem, da vsebuje poleg 2D še 3D pogled (sliki 7.3 in 7.4) in ima manjši naboru funkcionalnosti.



Slika 7.3: Jose 2D



Slika 7.4: Jose 3D

7.4 Testiranje

Implementiran program smo tudi testirali. Testirali smo ga s pomočjo testnih knjižnic ter v igri proti drugim šahovskim programom in amaterskim igralcem šaha. Testiranje smo izvajali na računalniku z procesorjem Pentium 4 2.8 GHz in 512 MB pomnilnika. Testni program pa je uporabljal alfa-beta algoritem, klestenje z ničelno potezo, aspiracijsko iskanje, iskanje mirovanja, transpozicijsko tabelo, ubijalsko hevristiko in bitno predstavitev igre. V primeru testnih knjižnic je program dosegel 53.3 % uspešnost. Podrobnejši opis testiranja prikazuje naslednja tabela.

Testna knjižnica	**Rezultat**	**Čas**	**Uspešnost**
Encyclopedia of Chess Middlegames	273/879	5 sec.	31.1%
Win at Chess	222/300	5 sec.	74.0%
1001 Wining Chess Sacrifices	666/1001	5 sec.	66.5%
Skupaj	**1161/2180**		**53.3%**

Iz rezultatov vidimo, da je naš program relativno uspešen v primeru pozicij, ki vodijo k zmagi. V primeru teh pozicij je naš program dosegel 68.3 % uspešnost. Tako lahko sklepamo, da naš program vsebuje relativno zmogljive iskalne algoritme in predstavitev igre. Nekoliko slabše rezultate pa smo dosegli pri pozicijah, ki se pojavljajo v sredini igre. Tukaj smo dosegli le 31.1% uspešnosti. Iz teh razultatov lahko sklepamo, da naš program vsebuje ocenitveno funkcijo, ki vsebuje premalo šahovskega znanja. Tako z dodajanjem določenega znanja ocenitveni funkciji, lahko zmogljivost našega programa še povečamo.

Proti človeku se je naš program izkazal bolje, kot v primeru reševanja testnih pozicij. V primeru iger, kjer smo programu omejili čas, za igranje ene poteze na eno minuto, je dosegel približno 60% uspešnost. V primeru hitropoteznega šaha pa se je njegova uspešnost povečala na 80%.

V primerjavi z trenutno najboljšimi šahovskimi programi, kot so npr. Fritz, Junior, GNUChess, itd. smo z našim programom uspeli doseči le remi. Razlog temu pa ja, da se ti programi razvijajo že več let in implementirani so v programskem jeziku C. Iz teh razlogov so ti programi bolj optimizirani in dosegajo večje globine iskanja.

Iz dobljenih rezultatov lahko vidimo, da je implementiran program relativno dober. Izboljšamo pa ga lahko tako, da ocenitveni funkciji dodamo še določena znanja. Dodatno pa lahko s pomočjo določene metode strojnega učenja poskusimo izboljšati (prilagoditi našemu programu) vrednosti parametrov v ocenitveni funkciji. Poleg izboljševanja ocenitvene funkcije lahko program izboljšamo še s pomočjo uporabe določenih algoritmo in zgodovinske hevristike.

8. Zaključek

V diplomski nalogi smo se seznanili z implementacijo računalniškega šaha. Spoznali smo sestavne dele šahovskih programov in njihove zmogljivosti. Osnovni sestavni deli šahovskih programov so: predstavitev igre, iskalni algoritmi, generator potez, ocenitvena funkcija, transpozicijska tabela, UCI vmesnik in grafični uporabniški vmesnik.

V okviru predstavitev igre smo spoznali več različnih predstavitev. Najboljša in najkompleksnejša predstavitev je bitna predstavitev. Ker uporablja bitne operacije, je znatno hitrejša, v primerjavi z katerokoli drugo predstavitvijo. Predstavitev igre uporabljamo pri iskalnih algoritmih. Osnovni algoritem, na katerem temeljijo šahovski programi, je alfa-beta algoritem. Za časovno omejena iskanja pa se uporabljata iterativno poglabljanje in aspiracijsko iskanje. Izboljšavo alfa-beta algoritma predstavljata NegaScout in MTD(f) algoritma. Prvi temelji na iskanju glavne variante, drugi pa na iskanju z minimalnim oknom.

Pri vseh algoritmih, ki temeljijo na alfa-beta algoritmu, je potrebno uporabiti še zbiranje glavne variante in iskanje mirovanja. Zbiranje glavne variante omogoča, poleg podajanja ocenitve, še podajane izbrane glavne variante. Iskanje miroavna pa odpravlja učinek obzorja oz. nadaljuje iskanje do statičnih pozicij, ki jih nato oceni s pomočjo ocenitvene funkcija. Za doseganje večjih globin iskanje, se v algoritmih uporablja še transpozicijska tabela, ubijalska hevristika in zgodovinska hevristika. Transpozicijska tabela odpravlja redudantna iskanja. Zgodovinska in ubijalska hevristika pa se uporablja za generiranje "dobrega" zaporedja potez.

Dodatno pa lahko uporabimo še selektivne algoritme. Primer selektivnega algoritma je iskanje z ničelno potezo. To iskanje dodatno klesit iskalno drevo ter tako dosega večjo globino iskanja. Zaradi klestenja z ničelno potezo, pa algoritem v določenih pozicijah odpove. Tako ta algoritem vsebuje nestabilna iskanja. Število pozicij, v katerih pride do nestabilnega isknja pa lahko zmanjšamo s omočjo adaptivnega faktorja reduciranja

in verifikacije ničelne poteze.

Znanje šahovskega programa predstavlja ocenitvena funkcija. Ta funkcija je lahko, glede na vsebovano znanje, bolj ali manj kompleksan. Bolj kompleksne funkcije vsebujejo več znanja, so pa zato časovno bolj zahtevne. Zato je potrebno, za zmogljiv šahovski program, najti pravo razmerje med hitrostjo in znanjem, ki ga ocenitvena funkcija vsebuje.

Na osnovi predstavljenih setavnih delov šahovskih programov, smo implementirali referenčni šahovski program. Program smo načrtovali tako, da omogoča dodajanje in testiranje novih iskalnih algoritmv in predstavitev igre. Na osnovi tega smo implementirali bitno predstavitev igre in različne iskalne algoritme. Ta implementacija se je izkazala kot zelo zahtevna. Implementirali smo še UCI vmesnik, ki nam je omogočil povezovanje našega programa z grafičnimi vmesniki. Nato smo program še testirali. Ugotovili smo, da je program relativno zmogljiv, ima pa premalo vgrajenega šahovskega znanja v ocenitveni funkciji.

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] Erest A. Heinz: How Drak Thought PlaysChess, Institute for Program Structure and Data Organization (IPD), ICCA Jurnal 20(3), 166-176, September 1997.

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

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

[5] Omid David Tabibi, Natan S. Netanyahu: Verified Null-Move Pruning, October 2002.

[6] 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.

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

[8] Strategy and board game programming,
dostopno na naslovu http://www.ics.uci.edu/ eppstein/180a/w99.html

[9] A short history of computer chess,
dostopno na naslovu http://www.chessbase.com/columns/column.asp?pid=102

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

[11] Marjan Mernik,Matej Črepinšek, Viljem Žumer: Evolucijski algoritmi, Fakulteta za elektrotehniko, računalništvo in informatiko, Inštitut za računalništvo, Maribor 2003.

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

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

[14] Grafučni uporabniški vmesnik Arena,
dostopno na naslovu http://www.playwitharena.com/

[15] Grafični uporabniški vmesnik Jose,
dostopno na naslovu http://jose-chess.sourceforge.net/