Aki Seppälä

TEKOÄLYN OHJELMOINTI SHAKKIPELILLE

Tietotekniikan

Pro gradu -tutkielma

29.12.2004

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

URN:NBN:fi:jyu-20054

Tekijä: Aki Seppälä

Yhteystiedot: Sähköposti aiseppal@cc.jyu.fi ja puh. 044 – 0665 997

Työn nimi: Tekoälyn ohjelmointi shakkipelille

Title in English: Programming Artificial Intelligence for the Game of Chess

Työ: Pro gradu –tutkielma

Sivumäärä: 128

Linja: Ohjelmistotekniikka

Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos

Avainsanat: Tekoäly, shakki, ohjelmointi, peliteoria

Keywords: Artificial intelligence, chess, programming, game theory

Tiivistelmä: Tässä opinnäytteessä tutkitaan tekoälyn shakkiohjelmointitekniikoita. Lisäksi toteutetaan perustekniikoilla shakkia pelaava ohjelma.

Abstract: This thesis examines program techniques how to create artificial intelligence for chess. In addition we program basic chess program using basic techniques.

SISÄLTÖ

1 JOHDANTO **5**

2 LAUDAN REPRESENTAATIO JA SALLITTUJEN SIIRTOJEN TUOTTAMINEN **7**

2.1 8X8 TAULUKKO 7
2.2 12X10 TAULUKKO 7
2.3 0X88 9
2.4 BITBOARD 10
2.5 FPGA 11

3 ASEMAN ARVIOINTI **12**

3.1 MATERIAALI 12
3.1.1 Perusmateriaalifunktio *12*
3.1.2 Ehdollisesti nappuloiden vaihtamista suosiva materiaalifunktio *13*
3.2 LIIKKUVUUS 14
3.3 KEHITYS JA LINNOITUS 16
3.3.1 Yksinkertainen kehitysfunktio *17*
3.3.2 Kehittynyt kehitysfunktio *17*
3.4 PAINOSTUS VASTUSTAJAN KUNINGASTA KOHTAAN 21
3.4.1 Perushyökkäyskaavio *22*
3.4.2 Aggressiivinen hyökkäyskaavio *24*
3.5 SOTILASASEMA 25
3.6 KESKUSTAN HALLINTA 27
3.6.1 Peruskeskustakaavio *27*
3.6.2 Keskustaa reunoillakin suosiva kaavio *30*
3.6.3 Oma yhdistelmäkaavio *33*
3.6.4 Laudan toiselle puolelle painostava keskustakaavio *35*
3.7 KUNINKAAN TURVALLISUUS 36
3.8 NAPPULOIHIN KOHDISTUVAT UHKAUKSET JA NIIDEN SUOJAUKSET 40
3.9 LOPPUPELIPIIRTEET 45
3.9.1 Yleinen loppupeli ja sotilasloppupeli *45*
3.9.2 Matin pakottaminen upseereilla *47*
3.10 MUITA OSA-ALUEITA 49
3.11 OSA-ALUEIDEN VALINTA JA PAINOKERTOIMIEN MÄÄRITYS 50
3.11.1 Arvaaminen *51*
3.11.2 Systemaattinen testaus *52*
3.11.3 Automaattinen määritys *52*

4 HAKUTEKNIIKAT **54**

4.1 ESIHAKU TAULUKOISTA 54
4.1.1 Avauskirjasto *54*
4.1.2 Loppupelitaulukot *56*
4.2 HAKUALGORITMIT 56
4.2.1 Minimax alfa-beta *61*
4.2.2 Heuristinen haku *64*
4.3 KILLER HEURISTIIKKA 64
4.4 HORISONTTIONGELMA 67
4.4.1 Valikoiva jatkolaskenta *70*
4.4.2 Pysyvät muistitaulukot *70*
4.4.3 Omia ideoita *71*

5 SOVELLUSESIMERKKI **73**

5.1 LÄHTÖKOHDAT 73
5.2 TOTEUTUS 73
5.2.1 Laudan representaatio ja sallittujen siirtojen tuottaminen *73*
5.2.2 Perusmateriaalifunktio *75*

5.2.3 Välivaiheen hakualgoritmi ... *75*
5.2.4 Laillisten siirtojen tuottaminen ... *77*
5.2.5 Laillisuuden huomioiva hakualgoritmi ... *78*
5.2.6 Arvotusfunktion osat ... *79*
5.2.7 Kokonaisuus ... *80*
5.3 TESTAUS ... 81
5.3.1 Pelaaminen ... *82*
5.3.2 Oman mattitehtävän ratkaisuttaminen ... *82*
5.3.3 Peliaseman analysointi ... *84*
5.4.4 Oman uuden lineaarisen hakustrategian testaus ... *85*
5.3.5 Johtopäätökset ... *85*
5.4 PARANNUSIDEAT ... 86

6 YHTEENVETO ... 88

LÄHTEET ... 90

LIITTEET ... 93

LIITE 1. SHAKKIOHJELMAN KOODI ... 93
LIITE 2. MATTITEHTÄVIEN RATKAISUT ... 125

1 JOHDANTO

Tekoälytutkimus antaa lähtökohdan keinotekoisen ihmisen rakentamiseksi. Arkipäiväisten luonnollisten tehtävien kuten näkemisen ja liikkumisen lisäksi tekoälyssä tutkitaan ihmisen itsensä kehittämiä, älykkyyttä vaativia tehtäviä. Eräs perinteinen tutkimuskenttä on pelit ja erityisesti shakki. Se on ikivanha lautapeli, jota pelaa kaksi pelaajaa siirtämällä nappuloita tiettyjen sääntöjen mukaan vuorotellen päämääränään tehdä vastapelaajan kuninkaasta matti. Shakin ohjelmointia koneelle on tutkittu 1950-luvulta lähtien, jolloin Claude Shannon kirjoitti aiheesta ensimmäisen artikkelin. Shakki soveltuu erityisen hyvin tekoälytutkimuksen kohteeksi [Shannon, 1950 ja Winston ym., 1979 s.4]:

- Shakin säännöt ovat selkeät ja niitä on vähän.
- Shakki ei ole triviaali ratkaistava, eikä toisaalta liian vaikeakaan tyydyttävän ratkaisun saamiseksi.
- Ohjelma voi tietää paljon shakista. Sillä voi olla esimerkiksi avauskirjasto avaussiirroista.

Shakkiohjelmista on hyötyä harjoitusvastustajina ja lisäksi ne ovat verrattomia analyysin apuvälineitä.

Tämän tutkimuksen tarkoitus on selvittää, mitä eri tekniikoita tekoälyn rakentamiseksi shakkipelille on olemassa ja miten käytännössä ohjelmoidaan toimiva kokonaisuus. Tutkimus jakaantuu teoriaosaan ja empiiriseen osaan. Teoriaosassa tutkitaan kirjallisuudesta mitä tekniikoita on olemassa ja empiirisessä osassa sovelletaan ja testataan näitä tekniikoita tekemällä oma ohjelma.

Shakkitekoälyn rakentaminen voidaan jakaa kolmeen osaan, jotka on eroteltu omiksi luvuikseen. Luvussa 2 käsitellään eri tapoja miten kone saadaan hahmottamaan lauta ja nappulat ja miten asemasta tuotetaan kaikki sallitut siirrot. Näin syntyneiden uusien asemien arviointiin käytetään arvotusfunktiota (evaluation function), joka antaa aseman hyvyydestä numeerisen arvion ja tätä käsitellään luvussa 3. Jotta ohjelma näkisi pidemmälle kuin yhden puolisiirron päähän, näistä asemista tuotetaan edelleen uusia asemia. Tuloksena on pelipuu, josta hakualgoritmi etsii

molemmin puolin parhaan mahdollisen jatkon. Tätä käsitellään luvussa 4.

Empiirisessä osassa, luvussa 5, kerrotaan sovellusesimerkin toteuttamisesta. Tarkoituksena on koodata oma shakkia pelaava ohjelma, joka pystyy voittamaan tekijänsä. Luvussa 6 arvioidaan tutkielmaa ja pohditaan mihin kaikkeen tutkielman tuloksia voisi soveltaa.

2 LAUDAN REPRESENTAATIO JA SALLITTUJEN SIIRTOJEN TUOTTAMINEN

Valmiiseen shakkiohjelmaan kuuluu kolme osaa ja siirtojen tuottaminen on ensimmäinen toteutettava osa. Tässä luvussa käsitellään ja vertaillaan eri tapoja miten kone saadaan hahmottamaan lauta ja nappulat ja miten asemasta tuotetaan kaikki sallitut siirrot. Siirtojen tuottaminen on merkittävä osa-alue myös tehokkuuden kannalta, sillä ohjelman miettiessä siirtoaan CPU-ajasta suuri osa kuluu pelkästään siirtojen tuottamiseen.

Käytännön ohjelmointiin hyvin sopivia tekniikoita ovat luvuissa 2.2, 2.3 ja 2.4 esitetyt menetelmät. Aloittelevalle ohjelmoijalle sopii hyvin luvun 2.2 konkreettinen menetelmä, kun taas ammattimaiseen ohjelmointiin paras lienee luvun 2.4 menetelmä, joka on monimutkainen toteuttaa, mutta erittäin tehokas.

2.1 8x8 taulukko

Yksinkertaisin ja ensimmäisenä mieleen tuleva tapa laudan koodaamiseen koneelle on 8x8 taulukko. Taulukon arvoina ovat vastaavan ruudun nappulat tai ruudun ollessa tyhjä erityinen tyhjä-arvo. Mahdolliset arvot ovat siis vS (valkea sotilas), vL (valkea lähetti), vR (valkea ratsu), vD (valkea daami) , vK (valkea kuningas), mS (musta sotilas), mL (musta lähetti), mR (musta ratsu), mD (musta daami), mK (musta kuningas) ja tyhjä-arvo. Tämän tavan etuna on se, että materiaalin laskeminen on helppoa. Käydään vain silmukassa kaikki ruudut läpi ja lasketaan arvot yhteen.

Ongelmiin törmätään kun pitäisi tuottaa sallitut siirrot. Käytännössä on hyvin hankalaa havaita milloin ollaan menty ulos laudalta. Tähän tarvittaisiin monimutkaisia testejä ja siksipä tätä tapaa ei juurikaan käytetä.

2.2 12x10 taulukko

Yleisesti käytetty perinteinen tapa laudan koodaamiselle on 12x10 taulukko. Laudan yläpuolella ja

alapuolella on ylimääräiset kaksi riviä ja kummallakin sivulla yksi ylimääräinen sarake (kuva 1). Shannonin ehdotuksen [Shannon, 1950] mukaisesti nappulat esitetään lukuina. 1 = valkea sotilas, 2 = valkea ratsu, 3 = valkea lähetti, 4 = valkea torni, 5 = valkea daami, 6 = valkea kuningas ja -1 = musta sotilas, -2 = musta ratsu, -3 = musta lähetti, -5 = musta daami, -6 = musta kuningas sekä tyhjälle ruudulle erityisarvo, joka on yleensä 7 tai 99 (kuva 2).

111	112	113	114	115	116	117	118	119	120
101	102	103	104	105	106	107	108	109	110
91	92	93	94	95	96	97	98	99	100
81	82	83	84	85					
71	72								
61	62								
51	52	53							
41	42	43	44						
31	32	33	34	35	36	37	38	39	40
21	22	23	24	25	26	27	28	29	30
11	12	13	14	15	16	17	18	19	20
1	2	3	4	5	6	7	8	9	10

Kuva 1 [Kozdrowicki, 1973]

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

Kuva 2 [Kozdrowicki, 1973]

Tämän tavan etuna edelliseen on, että nyt siirtojen tuottaminen on helpompaa, kun laudan rajat voidaan havaita. Mahdolliset siirrot voidaan laskea matemaattisina laskutoimituksina. Tietoina tarvitaan ruutu missä nappula on ja miten se liikkuu. Otetaan esimerkkinä ratsu laudan ruudussa a1 eli taulukossa ruudussa 22. Mahdolliset ratsun siirrot lasketaan kaavalla +8, +19, +21, +12, -8, -19, -21 ja -12. Tässä tapauksessa ensimmäinen siirtovaihtoehto on ruutu 22 + 8 = 30, mutta se on oikean laudan ulkopuolella, joten siirto ei ole mahdollinen. Seuraava yritys on 22 + 19 = 41, mutta sekin on laudan ulkopuolella. Kolmas yritys on mahdollinen eli 22+21 = 43, joka oikealla laudalla on ruutu b3. Neljäs yritys on myös laudalla eli 22+12 = 34, joka vastaa ruutua c2. Loput siirtovaihtoehtoruudut ovat 14, 3, 1 ja 10, jotka ovat kaikki oikean laudan ulkopuolella, mutta 12x10 taulukon sisällä. Näin tuli samalla todettua, että yksi sarake laudan sivuilla riittää eikä tarvita kahta, sillä edes ratsu nurkassa ei pysty hyppäämään taulukon ulkopuolelle. Siirron pysyminen laudalla ei kuitenkaan vielä takaa siirron laillisuutta. Ensimmäisenä täytyy tarkistaa, ettei ruudussa ole jo omanvärinen nappula. Tämä testi käy helposti, sillä valkeat nappulat ovat kaikki positiivisia numeroita ja mustat negatiivisia. Erikoissiirrot, kuten ohestalyönti ja linnoitus, tekevät siirtogeneraattorista hieman monimutkaisemman, mutta kokonaisuus säilyy tyydyttävän tehokkaana.

2.3 0x88

Modernimpi menetelmä on nk. 0x88 systeemi [Moreland]. Tässä "oikean" laudan oikealla puolella on toinen lauta, jolloin ruutuja on yhteensä 128. Tämän käyttämättömän lisälaudan ansiosta voidaan tehokkaasti yhdellä lauseella testata ollaanko laudalla vai ei. Katso kuva 3.

112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Kuva 3. Vasemmalla puolella on oikea lauta ja oikealla puolella 0x88-menetelmän tarvitsema lisälauta.

Lause yhdistää kaksi testiä, 0x80 testin, joka testaa ollaanko kaksoislaudan 128 ruudun alueella ja 0x08 testin, joka testaa ollaanko oikean laudan alueella. Testilause on seuraavanlainen:

i laillinen jos ja vain jos (i&0x88)==0.

Jokainen ruutu käsitetään 8 bitin sanana, joista ensimmäiset 4 bittiä kertovat rivin ja jälkimmäiset 4 bittiä sarakkeen. Merkittävä on neljän viimeisen bitin ensimmäinen bitti. Jos tämä bitti on 1, niin ollaan oikean laudan ulkopuolella. Otetaan selvennyksen vuoksi pari esimerkkiä. Olkoon ruudun numero 52. 52 on binäärilukuna 0011 **0**100. Rivi 0011 on kymmenjärjestelmän lukuna 3 ja sarake 0100 on 4. Tässä riittää kuitenkin tietää vain tärkeän 4. viimeisen bitin arvo, joka on 0. Tästä tiedetään, että ollaan oikealla laudalla. Tässä tapauksessa ollaan ruudussa e4. Otetaan toinen esimerkki. Jos oltaisiin ruudussa 55 ja siirryttäisiin yksi oikealle ruutuun 56, joka on binäärilukuna

0011 **1**000, niin huomattaisiin heti, että siirto on laiton, sillä 4. viimeisin bitti on 1.

Tämän tavan etuja ovat nopeus ja koodin selkeys.

2.4 Bitboard

Toinen moderni menetelmä on nk. Bitboard [Eppstein]. Se perustuu 64-bittisiin sanoihin. Ensimmäinen bitti vastaa ruutua a1 ja viimeinen bitti ruutua h8. Jokaiselle nappulatyypille on oma 64-bittinen sanansa. Jos nappula on ruudussa, bitin arvoksi tulee 1, muutoin 0. Esimerkiksi valkean sotilaat alkuasemassa ilmaistaisiin seuraavanlaisena bittikarttana:

00000000
00000000
00000000
00000000
00000000
00000000
11111111
00000000

Ja vastaavasti mustan sotilaiden sijainnin bittikartta näyttäisi tällaiselta:

00000000
11111111
00000000
00000000
00000000
00000000
00000000
00000000

Siis sekä valkeille että mustille nappulatyypeille tarvitaan oma bittikartta, joten kaikkiaan tarvitaan 12 64-bittistä sanaa laudan nappuloiden sijainnin määrittämiseen. Lisäksi voidaan määritellä muita hyödyllisiä sanoja, esimerkiksi ”kaikki valkean nappulat” ja ”kaikki mustan nappulat”. Tämä on

erityisen kätevää sallittujen siirtojen tuottamisen yhteydessä, jolloin riittää yhdistää nappulan mahdollisten siirtojen bittikartta omien nappuloiden bittikartan negaatioon.

Tämä menetelmä on monipuolinen ja tehokas ja soveltuu erinomaisen hyvin nykyaikaisiin 64-bittisiin prosessoreihin. Haittapuolena on lähinnä vain ohjelmoinnin monimutkaisuus.

2.5 FPGA

Supertietokoneissa, kuten Deep Blue:ssa, joka voitti Garry Kasparovin historiallisessa ottelussa vuonna 1997, käytetään vieläkin monimutkaisempaa tekniikkaa. Boule ja Zilic selostavat artikkelissaan [Boule ja Zilic, 2002] Deep Blue:n FPGA pohjaisen siirtogeneraattorin toimintaa. Keskeinen idea on, että jokaisella ruudulla on vastaanotin(receiver) ja lähetin(transmitter) ja jokainen ruutu kommunikoi vain naapuriruutujensa kanssa. Menetelmä on hyvin monimutkainen ja liian laaja tähän yhteyteen, joten ei ole tarkoituksenmukaista käsitellä sitä menetelmää tässä sen tarkemmin.

3 ASEMAN ARVIOINTI

Edellisesä luvussa esiteltiin miten asemasta tuotetaan lailliset siirrot. Syntyneiden uusien asemien arviointiin käytetään arvotusfunktiota (evaluation function), joka on keskeinen osa shakkiohjelmien tietämystä. Ohjelma, joka vain tuottaisi kaikki sallitut siirrot ja arvioisi kunkin näin syntyneen aseman, ilman että laskisi yhtään eteenpäin, pelaisi jo jonkunlaista shakkia. Tässä luvussa käsitellään tärkeimpiä arvotusfunktion sisältämiä asioita.

3.1 Materiaali

Tärkein asia, mikä arvotusfunktion on otettava huomioon, on ilman muuta materiaalitilanne.
Jos muut asiat ovat tasan, se pelaaja, joka on materiaalissa edellä, yleensä voittaa.

3.1.1 Perusmateriaalifunktio

Yksinkertaisin tapa laskea materiaali on käydä lauta läpi ja laskea sotilaan arvoina kummalla on materiaalia enemmän käyttäen seuraavaa perinteistä asteikkoa:

sotilas s (Pawn p) = 1
ratsu R (Knight N) = 3
lähetti L (Bishop B) = 3
torni T (Rook R) = 5
kuningatar D (Queen Q) = 9
kuningas K (King K) = hyvin suuri arvo, esim. 1000

Kuningas on korvaamaton, joten antamalla sille hyvin suuren arvon, kone välttää sen menettämistä viimeiseen asti. Lähettiä pidetään aavistuksen vahvempana kuin ratsua, joten lähetin arvo voisi olla myös 3,25 tai 3,5. Kuitenkin suljetuissa asemissa lähetti ei ole parempi vaan voi olla jopa huonompi, joten ylläoleva asteikko on tyydyttävän tarkka. Shannon esitti materiaalifunktion muodossa [Shannon, 1950]:

f(P) = 200(K-K') + 9(Q-Q') + 5(R-R') + 3(B-B' + N–N') + (P-P'),
missä K, Q, R, B, N ja P ovat valkean ja heittomerkillä varustetut mustan kuninkaiden, kuningattarien, tornien, lähettien, ratsujen ja sotilaiden lukumäärä.

3.1.2 Ehdollisesti nappuloiden vaihtamista suosiva materiaalifunktio

Vaihdoista saatava vaihtobonus ("trade down" bonus) kannustaa materiaalissa edellä olevaa osapuolta vaihtamaan nappuloita, mutta ei sotilaita [Frey, 1978, s.94]. Tätä menetelmää käyttää mm. CHESS 4.5-ohjelma. Nappuloilla on arvot sotilas = 100, ratsu = 325, lähetti = 350, torni = 500 ja kuningatar = 900, ja kokonaisfunktio on seuraavanlainen :

Total material advantage = minimum (3100, MS), missä
MS = minimum (2400, MD) + MD x PA x (8000 – MT)/(6400x(PA+1)).

MD on materiaaliero materiaalissa edellä olevan näkökulmasta, PA on materiaalissa edellä olevan sotilaiden lukumäärä ja MT on laudan yhteenlaskettu kokonaismateriaali.

Otetaan esimerkiksi tilanne, jossa valkealla ja mustalla on ollut kaikki nappulat laudalla, mutta musta on juuri menettänyt lähetin. MD lasketaan kohdassa 3.1.1 esitetyn materiaalitilannefunktion kaltaisesti eli MD = 8 x 100 + 2 x 325 + 2 x 350 + 2 x 500 + 1 x 900 - (8 x 100 + 2 x 325 + 1 x 350 + 2 x 500 + 1 x 900) = 350 ja kokonaismateriaali MT = 8 x 100 + 2 x 325 + 2 x 350 + 2 x 500 + 1 x 900 + 8 x 100 + 2 x 325 + 1 x 350 + 2 x 500 + 1 x 900 = 7750. PA = 8, koska valkealla on kaikki sotilaat vielä laudalla. Seuraavaksi lasketaan MS. MS = minimum (2400, 350) + 350 x 8 (8000-7750)/(6400*(8+1)) = 362. Edelleen minimum (3100, 362) = 362, joten kokonaisetu on 362. Oletetaan että nyt materiaalissa edellä olevalla olisi mahdollisuus vaihtaa ratsu ratsuun tai sotilas sotilaaseen. Miten tilanne muuttuisi kussakin vaihtoehdossa ? Tutkitaan ensin ratsujen vaihtaminen. MT olisi nyt 8 x 100 + 1 x 325 + 2 x 350 + 2 x 500 + 1 x 900 + 8 x 100 + 1 x 325 + 1 x 350 + 2 x 500 + 1 x 900 = 7100 ja MD ja PA pysyvät samoina. MS = minimum (2400, 350) + 350 x 8 (8000-7100)/(6400*(8+1)) = 394. 394 – 362 = 32, joten kokonaisetu kasvaisi noin sotilaan kolmanneksella. Jos ratsujen vaihdon sijaan materiaalissa edellä oleva vaihtaisi sotilaan sotilaaseen, niin tällöin MT = 7 x 100 + 2 x 325 + 2 x 350 + 2 x 500 + 1 x 900 + 7 x 100 + 2 x 325 + 1 x 350 + 2 x 500 + 1 x 900 = 7550, PA on nyt 7 ja MD on edelleen 350. MS = minimum (2400, 350) + 350 x 7 x (8000-7550)/(6400*(7+1)) = 371. 371 – 362 = 9, joten kokonaisetu kasvaisi vain aavistuksen. Ratsujen vaihdosta sai etua 32 pistettä, mutta sotilaiden vaihdosta vain 9 pistettä, joten materiaalissa

edellä olevan kokonaisetu kasvaa huomattavasti enemmän, kun vaihtaa ratsun ratsuun verrattuna siihen, että vaihtaisi sotilaan sotilaaseen.

3.2 Liikkuvuus

Heti materiaalin jälkeen toiseksi tärkein asia on liikkuvuus. Nappuloiden kokonaisliikkuvuus lasketaan niiden ruutujen määränä, joihin nappulat pääsevät siirtymään. Tämä yksinkertainen lasku kuvaa hyvin nappuloiden liikkumisvapautta. Jos tähän yhdistetään edellisen luvun 3.1. materiaalifunktio, saadaan arvotusfunktioksi:

aseman arvo = materiaaliero + (0,1 x liikkuvuusero)

0,1 on perinteinen kerroin, jota suositteli jo Shannon [Shannon, 1950]. Vaikka 0,1 toimiikin hyvin joissakin asemissa, se on lähinnä kertoimen yläraja. Liikkuvuuden pätevä painokerroin on suunnilleen väliltä 0,03 – 0,1 ja itse asiassa lähempänä arvoa 0,03, koska ohjelmat tekevät yleensä parempaa tulosta painokertoimella 0,03 kuin 0,1. Tämä yksinkertainen funktio antaa jo melko hyvän arvion asemasta. Varhaiset shakkiohjelmat, kuten LOS ALAMOS vuodelta 1956, käyttivät tällaista arvotusfunktiota [Newell ym., 1958]. Otetaan esimerkiksi aloitusasema. Valkealla ja mustalla on kummallakin 20 erilaista jatkomahdollisuutta. Jos valkea avaa 1.a3, niin jatkomahdollisuuksia on vain 19 ja valkean kokonaisliikkuvuus on siis 19. Mustan kokonaisliikkuvuus on edelleen 20. Funktio antaa nyt tuloksen 0,1x(19-20) = -0,1 eli valkea on heti menettänyt avausetunsa. Jos taas valkea avaa 1.e4, niin jatkomahdollisuuksia on 30 ja aseman arvo on 0,1x(30-20) = 1. Siirrolla 1.d4 jatkomahdollisuuksia olisi 29 ja aseman arvo 0,9. Hyvät pelaajat avaavatkin pelin usein juuri siirroilla 1.e4 tai 1.d4, jotka antavat erinomaiset jatkomahdollisuudet säilyttäen valkean avausedun Lisäksi on osoitettu, että voittavan pelaajan nappuloiden liikkumismahdollisuudet kasvavat tasaisesti pelin edetessä suhteessa häviävän pelaajan nappuloiden liikkuvuuteen [Slater, 1950 ja Levy, 1984, s. 8-11].

Otetaan esimerkiksi vielä yksi suosittu avausmuunnelma, joka kuvaa hyvin liikkuvuuden merkitystä.

1.d4 Rf6
2.c4 g6
3.Rc3 d5

4.cxd5 Rxd5

5.e4 Rxc3

6.bxc3 Lg7

7.Lc4 0-0

8.Re2 c5

9.0-0 Rc6

10.Le3 Lg4

11.f3 cxd4

12.cxd4 Ra5

13.Ld3 Le6

Katso kuvio 1.

Kuvio 1

Materiaali on tasan, mutta musta painostaa hieman d4-sotilasta ja ruutua c4.

Valkean liikkuvuus on 35.

Mustan liikkuvuus on 38.

Aseman arvo = 0,1 x (35-38) = - 0,3

14.d5 Terävä uhraussiirto, jota ovat käyttäneet valkeilla mm. sellaiset huippupelaajat kuin suurmestari John Nunn ja moninkertainen suomenmestari Joose Norri

14. - Lxa1

15.Dxa1 f6

Katso kuvio 2.

Kuvio 2

Valkea on uhrannut materiaalissa kvaliteetin, mutta on saanut korvaukseksi paremman liikkuvuuden.

Valkean liikkuvuus on 46

Mustan liikkuvuus on 32.

Aseman arvo = (3 – 5) + 0,1 x (46 – 32) = -0,3

Ei ole täysin selvää korvaako liikkuvuus tässä täysin kaksi sotilasta. NCO[Nunn ym., 1999, s.434] antaa asema-arvioksi epämääräisen aseman. Mutta koska monet vahvat pelaajat ovat lähteneet tähän muunnelmaan valkeilla, materiaalin vaihdon liikkuvuuteen täytyy olla kannattavaa tai ainakin lähes kannattavaa. Arvotusfunktio liikkuvuuden painokertoimella 0,1 osaisi myös arvostaa siirtoa 14.d5. Tämän liikkuvuuden painokertoimen täsmällinen määrittäminen on yksi tietokoneshakin mielenkiintoisimpia asioita. Jos kerroin on liian iso, ohjelma voi uhrata paljon materiaalia saadakseen vain vähän etua liikkuvuudessa. Jos taas kerroin on liian pieni, ohjelma ei koskaan yritä parantaa asemaansa pienelläkään materiaalin kustannuksella.

3.3 Kehitys ja linnoitus

Pelin alussa upseerit sijaitsevat takarivillä. Heti pelin alussa pyritään kehittämään upseereita, etenkin kevyitä upseereita eli ratsuja ja lähettejä, keskemmälle lautaa, jossa ne vaikuttavat

tehokkaammin pelin kulkuun.

3.3.1 Yksinkertainen kehitysfunktio

Kehitys voidaan yksinkertaisimmillaan laskea alkuasemastaan lähteneiden (kevyiden) upseereiden lukumäärän perusteella [Hyvönen, 1993, s.179]. Lisäksi on olemassa sääntö, että kolme tempoa eli kolme kehitettyä nappulaa vastaa arvoltaan suunnilleen sotilasta [Levy, 1984, s.14]. Kun nämä kaksi seikkaa yhdistetään, saadaan seuraavanlainen kehitysfunktio:

kehitys = kevyiden upseereiden kehitysero / 3

Täsmällisesti ilmaistuna
kevyiden upseereiden kehitysero = niiden valkeiden upseereiden, jotka **eivät** ole alkuperäisillä paikoillaan, lukumäärä – niiden mustien upseereiden, jotka **eivät** ole alkuperäisillä paikoillaan, lukumäärä.

Määritelmä 'liikkunut alkuperäiseltä paikaltaan' ei olisi tarkka, koska joskus upseeri joutuu palaamaan takaisin alkuperäiselle paikalleen.

3.3.2 Kehittynyt kehitysfunktio

Kevyiden upseereiden kehitys on kuitenkin vasta ensimmäinen osa kehityksestä. Yleisenä ohjenuorana shakissa on, että ensin kehitetään kevyet upseerit, sitten linnoitutaan ja sen jälkeen kehitetään raskaat upseerit (tornit ja kuningatar). Lisäksi kehityksessä pitäisi ottaa huomioon, että jos vastustajalla on kuningatar laudalla, linnoittuminen on lähes välttämätöntä, mutta jos vastustajan kuningatar on vaihtunut pois, kuninkaan jääminen keskustaan ei ole niin vaarallista. Näihin seikkoihin perustuen David Levy ehdottaa seuraavanlaista kehitysfunktiota [Levy,1984, s. 14] :

kehitys = D/3 – U/4 – (k x C), missä

D = niiden kevyiden upseereiden, jotka eivät ole alkuperäisillä paikoillaan, lukumäärä
U = 0, jos kuningatar ei ole liikkunut tai on lyöty pois
= kehittämättömien nappuloiden (kevyiden ja raskaiden) lukumäärä, jos kuningatar on liikkunut

ja on edelleen laudalla

C = 2, jos vastustajan kuningatar on vielä laudalla

= (1 – P/4), jos vastustajan kuningatar ei ole enää laudalla, missä P on vastustajalta lyötyjen kevyiden ja raskaiden upseereiden lukumäärä

k = 0, jos pelaaja on linnoittautunut

= 1/3, jos pelaaja on menettänyt daamipuolen linnoitusoikeuden

= 2/3, jos pelaaja on menettänyt kuningaslaidan linnoitusoikeuden

= 1, jos pelaaja on menettänyt kaikki linnoitusoikeudet

Funktio on looginen. D/3-termi toimii kuten kohdassa 3.3.1.

Jos pelaajan kuningatar on vielä laudalla, mutta on jo liikkunut, niin silloin pelaajaa rangaistaan U/4-termin avulla jokaisesta kehittämättä olevasta upseerista. Tämä kannustaa kehittämään muut nappulat ennen kuningatarta ja koska linnoittaminen itseasiassa kehittää sekä tornin että kuninkaan, se myös kannustaa linnoittamaan ennen kuningattaren siirtämistä.

K x C -termi rankaisee pelaajaa menetetyistä linnoitusoikeuksista. Pahinta olisi menettää kaikki linnoitusoikeudet, jolloin pelaajaa sakotetaan maksimiarvolla 1. Seuraavaksi pahinta olisi menettää linnoitusoikeudet kuningaslaidalle, koska kuningaslaita on paljon yleisempi linnoittumispuoli kuin daamipuoli ja kuningas on yleensä paremmin turvassa kuningaslaidalla kuin daamipuolella. Linnoitusoikeuden menettämisen vakuvuus riippuu siitä, onko vastustajan kuningatar vielä laudalla. Jos se on vielä laudalla, se voi johtaa pahoihin vaikeuksiin, joten C:n arvoksi asetetaan 2. Jos vastustajan kuningatar on lyöty pois, niin silloin tämä kehityksen osa ei ole niin tärkeä ja C saa arvoksi 1-P/4, missä P on vastustajalta lyötyjen upseereiden määrä. P:n arvo voi kääntää C:n arvon negatiiviseksi, jos vastustajalla ei ole enenpää kuin yksi torni tai yksi kevyt upseeri laudalla. Tämä kannustaa linnoittautumaan, jos peli on jo loppupelissä, mutta kuningas on edelleen alkuperäisellä paikallaan.

Tarkastellaan funktion toimivuutta terävässä kuningasgambiitin Muzio-muunnelmassa. Kyseisessä muunnelmassa valkea uhraa ratsun ja saa korvaukseksi merkittävän kehitysedun ja liikkuvuusedun. Valkea toivoo uhrauksellaan saavuttavansa aseman, missä hän voi käynnistää vaarallisen hyökkäyksen vastustajan kuningasta vastaan ja seurauksena vastustaja joutuu mattiin tai vaihtoehtoisesti saada takaisin kaikki tai osa uhratusta materiaalista samalla säilyttäen sama asemaetu. Olen joskus itsekin pelannut tätä muunnelmaa valkeilla ja on kiinnostavaa nähdä saadaanko kone tosiaan ymmärtämään tällaisia asemia kehittyneen kehitysfunktion avulla.

Muunnelma etenee seuraavasti:

1. e4 e5

2. f4 exf4

3. Rf3 g5

4. Lc4 g4

5. 0-0 !?

Muzion gambiitin avainsiirto. Valkea ei pelasta ratsuaan vaan jatkaa kehittämistä.

5. - gxf3

6.Dxd3 Df6

7.e5!

Tärkeä siirto. Valkea uhraa vielä sotilaankin avatakseen e-linjan tornilleen. Vastustajan kuninkaan ollessa keskustassa tämä on hyvä idea.

7. - Dxe5

8.d3 Lh6 9.Rc3 Re7 10.Ld2

On helppo nähdä, että valkea on kehityksessä edellä ja e-linjalta on tulossa hyökkäys. Shakin avauskirjojen "raamattu" NCO lopettaa analyysin jo tähän ja arvioi, että valkealla on riittävä korvaus uhratusta materiaalista [Nunn, 1999, s.296].

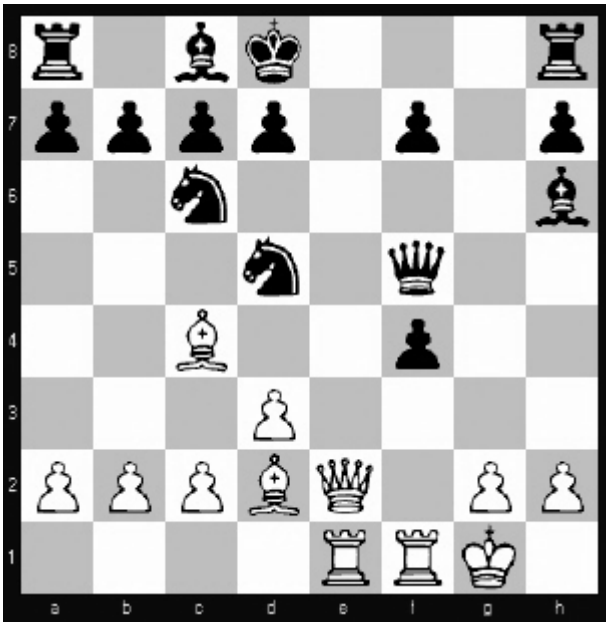
10. - Rc6

11.Tae1 Df5

12.Rd5 Kd8

13.De2

Katso kuvio 3.



Kuvio 3

Tämä on kriittinen asema koko muunnelman kannalta. Asemaa ovat analysoineet monet shakkimestarit ja tilannetta pidetään hyvin tasaisena. Ensi näkemältä 13. - Rxe5 14.Lxd5 Dxd5 näyttää hyvälle mustalle, mutta nyt siirto 15. Lc3! uhaten mattia alkaen Lf6+ on hyvin vahva. Niinpä mustan yleisin siirto tässä on 13. - De6. Valkea voi nyt halutessaan pakottaa mustan tasapeliin siirtojen toistolla jatkossa 14.Df3 Df5 15.De2 De6 16.Df3 Df5 ½-½ . Valkealla on myös 2 syvälle analysoitua mahdollisuutta jatkaa peliä, 14.Lc3 ja 14.Rxe7, joiden kummankin pitäisi johtaa tasapeliin kummankin puolen pelatessa tarkasti.

Tutkitaan nyt miten asemaa voisi lähestyä materiaalin, liikkuvuuden ja kehityksen kautta. Valkea on tappiolla materiaalissa ratsun ja sotilaan eli yhteensä 4 sotilasta. Valkean kokonaisliikkuvuus on 46 ja mustan 34 eli valkean liikkuvuusetu on 12. Jos käytetään kohdassa 3.2 mainittua yksinkertaista arvotusfunktiota, joka ottaa huomioon vain materiaalin ja liikkuvuuden, saadaan tulos

aseman arvo = -4 + 0,1 x 12 = -2,8

Tämän mukaan valkea olisi lähes 3 sotilasta tappiolla, mutta tutkitaan nyt miten aseman arvo muuttuu, kun siihen lisätään myös kehitysedun vaikutus.

Yritetään ensin soveltaa luvun 3.3.1 yksinkertaista kehitysfunktiota.

Sekä valkealla että mustalla on laudalla 3 kevyttä upseeria, jotka eivät ole alkuperäisillä paikoillaan, joten kummallekin tulee sama tulos:

kehitys = 3/3 = 1

Tämä yksinkertainen kehitysfunktio on ilmeisen epätarkka, koska sen mukaan valkealla ei olisi lainkaan kehitysetua.

Kokeillaan nyt kehittyneempää kehitysfunktiota.

D = 3 valkealle ja mustalle, koska kummallakin 3 kevyttä upseeria jotka eivät ole alkuperäisillä paikoillaan
U = 0 valkealle, koska valkean kuningatar on liikkunut, mutta kehittämättömiä nappuloita ei ole

= 3 mustalle, koska musta kuningatar on liikkunut ja kehittämättömiä nappuloita ovat 2 tornia ja 1 lähetti eli yhteensä 3 nappulaa

C = 2 valkealle ja mustalle, koska kummankin kuningatar on vielä laudalla

k = 0 valkealle, koska valkea on linnoittautunut

= 1 mustalle, koska musta ei ole linnoittautunut

Joten kaavasta

kehitys = D/3 – U/4 – (kxC)

saadaan nyt

valkean kehitys = 3/3 – 0/4 – (0 x 2) = 1

mustan kehitys = 3/3 – 3/3 – (1 x 2) = - 1,75

Valkean kehitysetu olisi siis 2,75 yksikköä. Yksi arvotusfunktion suunnittelun vaikeuksista on arvioida kunkin ominaisuuden paino suhteessa toisiinsa. David Levy arvioi, että hänen kehitysfunktionsa painoarvo suhteessa materiaalin perusyksikköön sotilaaseen on välillä 0,5 - 1. Näin ollen tähänastinen arvotusfunktio olisi välillä

aseman arvo = materiaali + 0,1 x liikkuvuus + 0,5 x kehitys

ja

aseman arvo = materiaali + 0,1 x liikkuvuus + kehitys

Ensimmäisessä tapauksessa saadaan -4 + 0,1 x 12 + 0,5 x 2,75 = -1,43 ja toisessa -4 + 0,1 x 12 + 2,75 = -0,05.

Asema-arvio -0,05 on ehkä liian positiivinen valkean kannalta, koska tässä vaiheessa arvotusfunktio ei ole vielä ottanut huomioon mustan sotilasheikkouksia eikä valkean painostusta mustan kuningasta kohtaan. Kun nämä seikat otetaan huomioon valkealla olisi jo etua, mutta teoreettisestihan aseman tulisi olla tasan. Asetetaan tässä tapauksessa kehityksen painoarvo lähelle alarajaansa, jolloin arvotusfunktioksi saadaan

aseman arvo = materiaali + 0,1x liikkuvuus + 0,6 x kehitys

3.4 Painostus vastustajan kuningasta kohtaan

Yksinkertaisin tapa arvioida kuningashyökkäyksen vahvuutta on antaa bonusta sitä enemmän mitä lähempänä vastustajan kuningasta omien nappuloiden hyökkäysruudut ovat.

3.4.1 Perushyökkäyskaavio

David Levy ehdottaa, että vastustajan kuninkaan ympärillä on seitsemän pisteen alue, josta jokainen oman nappulan uhka kartuttaa hyökkäyspisteitä seitsemällä pisteellä. Ruudun verran kaempana on kuuden pisteen vyöhyke ja niin edelleen [Levy, 1984, s.20].

Sovelletaan tätä mallia edellisessä alakohdassa 3.3.2 mainittuun asemaan (kuvio 3):

5 6 7 7 7 6 5 4
5 6 7 7 7 6 5 4
5 6 6 6 6 6 5 4
5 5 5 5 5 5 5 4
4 4 4 4 4 4 4 4
3 3 3 3 3 3 3 3
2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1

Kaavio 1

Lasketaan nyt arvot jokaisen valkean nappulan jokaisesta hyökkäyksestä.

Te1:	1+1+1+1+1+2 = 7
Tf1:	1+1+2+3+4 = 11
Kg1:	1+1+2+2+2 = 8
sa2:	3 (a2-sotilas voi liikkua ruutuihin a3 ja a4, mutta hyökkää vain ruutuun b3 ja vain hyökkäykset lasketaan)
sb2:	3+3 = 6
sc2:	3+3 = 6
Ld2:	1+1+3+4+5+3+4 = 21
De2:	1+1+1+2+2+3+3+4+5+6+7+3+4+4 = 46
sd3:	4+4 = 8
Lc4:	2+3+3+5+6+5 = 24
Rd5:	3+3+4+4+6+6+7+7 = 40

Valkean kokonaishyökkäys on 180.

Mustalle kuningasyökkäyslauta näyttäisi tältä:

1 1 1 1 1 1 1 1
2 2 2 2 2 2 2 2
2 3 3 3 3 3 3 3
2 3 4 4 4 4 4 4
2 3 4 5 5 5 5 5
2 3 4 5 6 6 6 6
2 3 4 5 6 7 7 7
2 3 4 5 6 7 7 7

Mustan hyökkäysruudut ovat:

Ta8:	1+2 = 3
Lc8:	2+2 = 4
Kd8:	1+1+2+2+2 = 8
Th8:	1+1+1+1+2 = 6
sa7:	3
sb7:	2+3 = 5
sc7:	3+3 = 6
sd7:	3+3 = 6
Re7:	1+1+3+3+4+4 = 16
sf7:	3+3 = 6
Rc6:	1+1+2+2+2+4+3+5 = 20
Lh6:	1+2+4+5 = 12
Df5:	2+3+2+3+2+3+4+4+4+4+5+5+5+5+6 = 57
sf4:	6+6 = 12

Mustan kokonaishyökkäys on 167.

Tämän mukaan valkealla olisi vain (180 – 167) 13 pistettä parempi hyökkäys. Otetaan lisäksi huomioon valkean tornin Te1 röntgenhyökkäys valkean daamin yli ruutuun e7. Tästä valkea saa lisäpisteitä 2+3+4+5+6+7 = 27. Samalla logiikalla myös alariviltä tulee muutama lisäpiste. Te1 hyökkää pisteeseen g1 Tf1:n läpi ja Tf1 hyökkää ruutuihin a1, b1, c1 ja d1 Te1:n läpi. Valkean

todellinen kuningashyökkäysvoima on siis 212 ja mustan edelleen 167. Nyt valkean etu on 45. Päivitetään edellisen alaluvun arvotusfunktiota ja otetaan mukaan myös painostus vastustajan kuningasta kohtaan:

aseman arvo = materiaali + 0,1x liikkuvuus + 0,75 x kehitys + 0,01 x kuningashyökkäys

Kun arvioidaan edelleen kuvion 3 asemaa, saadaan

aseman arvo = -4 + 0,1 x 12 + 0,6 x 2,75 + 0,01 x 45 = - 0,70

Vielä on otettava huomioon mustan heikko sotilasasema ja tämä tehdään seuraavassa alaluvussa 3.5.

3.4.2 Aggressiivinen hyökkäyskaavio

Jos halutaan tehdä ohjelmasta hyökkäävämpi, ei kannata laajentaa isoja pisteitä antavien hyökkäysruutujen aluetta eikä nostaa painotuskerrointa, vaan paras tapa on jyrkentää hyökkäyspisteiden nuosua vastustajan kuninkaan läheisyydessä ja kauempana kuninkaasta ei anneta lainkaan pisteitä. David Levy ehdottaa hyökkäävämmän mallin portaikoksi 1, 2, 4 ja 8 [Levy, 1984, s.22], jolloin edellisen kohdan hyökkäyslauta näyttäisi seuraavalta:

2 4 8 8 8 4 2 1
2 4 8 8 8 4 2 1
2 4 4 4 4 4 2 1
2 2 2 2 2 2 2 1
1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
Kaavio 2

3.5 Sotilasasema

Sotilasasema on tärkeä asia. Hyvät pelaajat kiinnittävät paljon huomiota siihen, että oma sotilasasema säilyisi eheänä. Sotilasasemaa heikentävät lähinnä kolme tekijää: kaksoissotilaat, eristetyt sotilaat ja takasotilaat. Kaksoissotilailla tarkoitetaan kahta samanväristä sotilasta peräkkäin samalla linjalla. Eristetty sotilas on sellainen sotilas, jonka kummallakaan puolella ei ole omia sotilaita ja täten sotilasta ei voi suojata mikään sotilas. Takasotilas on sotilas, joka on jäänyt jälkeen viereisistä omista sotilaista. Jos vastustajalla ei ole sotilasta takasotilaan linjalla, takasotilas on yleensä vakava heikkous. Erittäin paha heikkous on myös eristetty kaksoissotilas, eli kaksi omaa sotilasta peräkkäin eikä kummallaan viereisellä linjalla ole omaa sotilasta.

Vapaasotilas sen sijaan parantaa huomattavasti omaa asemaa. Vapaasotilas on sellainen sotilas, jonka linjalla eikä kummallakaan viereisellä linjalla ole vastustajan sotilasta, jolloin sotilas on potentiaalinen ehdokas korottumaan kuningattareksi.

Erilaisten sotilasheikkouksien suhde toisiinsa ja suhde materiaaliin ei ole triviaali. Ne vaihtelevat lähes joka ohjelmassa.

Ensimmäisen arvion määritteli Shannon [Shannon, 1950]:
sotilasasema = 0,5 x (D – D' + S – S' + I – I '), missä D, S ja I ovat valkeiden kaksoisssotilaiden, takasotilaiden ja eristettyjen sotilaiden lukumäärä

Tämä on kuitenkin melko epätarkka määritelmä nykyisin.

Tarkempi ja täsmällinen määritys on esimerkiksi IBM 7090-tietokoneelle kehitetyn shakkiohjelman sotilasasemaa arvioiva osa [Kotok, 1962]:

Heikennyksistä vähennetään pisteitä seuraavasti:

Eristetty sotilas	- 1
Takasotilas	- 5
Kaksoissotilas	- 3

Ja vahvuuksista saa pisteitä:

Vapaasotilas	+ 10

Nämä pisteet painotetaan 0,05-kertoimella suhteessa materiaaliin.

Hyvä on myös David Levyn antama määritelmä [Levy, 1984, s.26]:

Eristetty sotilas	- 4
Takasotilas	- 1
Kaksoissotilas	- 3

Levy ei anna vapaasotilaalla mitään arvoja, vaan toteaa, että vapaasotilaan arvo kasvaa sitä mukaa kun nappuloita vähenee ja sotilas lähenee korotusruutuaan. Hän ei myöskään anna minkäälaista arviota arvojensa suhteesta materiaaliin. Tutkin asiaa empiirisesti uusimmillla shakkiohjelmilla (mm. Rebel 10:lla) syöttämällä niille erilaisia sotilasemia ja tulin siihen tulokseen, että Davin Levyn mallissa painotuskerroin on noin 0,025.

Arvioidaan vielä kuvion 3 asema loppuun ottamalla mukaan myös sotilasasema. Mustalla on f linjalla eristetty kaksoissotilas. Se tarkoittaa, että mustaa rangaistaan kahdesti sekä kaksoissotilasvähennyksellä että eristetyn sotilaan vähennyksellä. Lisäksi mustalla on yksi eristetty sotilas h-linjalla, josta tulee vielä yksi vähennys. Itse asiassa tässä tapauksessa ei ole eroa käytetäänkö Levyn vai Kotokin kertoimia, sillä molemmat antavat saman tuloksen:

Kotok: 0,05 x (-3 – 1 - 3 -1 – 1) = - 0,45
Levy: 0,025 x (-3 - 4 – 3 – 4 – 4) = - 0,45

Nämä tulokset ovat hyvin lähellä todellisuutta, sillä myös uusimmat shakkiohjelmat antavat samansuuntaisia arvioita. Sen sijaan Shannonin kertoimilla tulokseksi tulisi useita sotilaita mikä olisi aivan liian paljon. Mustan asema siis heikentyy lähes puolella sotilaalla eli kääntäen valkean asema parantuu lähes puolella sotilaalla. Katsotaan nyt lopullinen arvotusfunktio

aseman arvo = materiaali + liikkuvuus + kehitys + kuningashyökkäys + sotilasasema
aseman arvo = -4 + 1,2 + 1,65 + 0,45 + 0,45 = - 0,25

Tulos on hyvin lähellä nollaa, kuten pitääkin. Jätetään nyt tämän aseman tutkiminen ja tutkitaan vielä joitakin muita aseman arviointiin liittyviä asioita, jotka tässä asemassa ovat suunnilleen tasapainossa eivätkä siis vaikuttaisi juurikaan arvotusfunktion tulokseen.

3.6 Keskustan hallinta

Keskustan hallinta on olennainen osa hyvän shakin pelaamisessa. Keskustaruuduista nappulat voivat hyökätä useampiin ruutuihin kuin jos ne olisivat laudan reunalla, joten on kannattavaa uhata keskustaruutuja, koska uhkauksien seurauksena oma nappula voi päästä asettumaan edulliselle keskustaruudulle.

Itsestään selvä menetelmä keskustan hallinnan arvioimiseksi on asettaa laudan ruuduille pisteitä siten, että keskemmällä oleville ruuduille laitetaan suuremmat arvot kuin reunemmassa oleville ruuduille. Ohjelma laskee tämän jälkeen jokaisen kuhunkin ruutuun kohdistuvan uhkauksen ja antaa uhkauksen suorittavalle puolelle ruudun arvon mukaisen bonuksen.

3.6.1 Peruskeskustakaavio

Seuraavassa on esitetty yksinkertainen keskustakaavio, joka on jo kohtalaisen toimiva [Levy, 1983, s.18]:

1 1 1 1 1 1 1 1
1 2 2 2 2 2 2 1
1 2 4 4 4 4 2 1
1 2 4 8 8 4 2 1
1 2 4 8 8 4 2 1
1 2 4 4 4 4 2 1
1 2 2 2 2 2 2 1
1 1 1 1 1 1 1 1

Kaavio 3

Katsotaan käytännön esimerkki keskustan hallinnan tärkeydestä. Mielestäni hyvin asiaa valaisee Blumenfeldin gambiitti, jossa musta uhraa sotilaan saadakseen keskustan haltuunsa. Vaikka pelaan itsekin kyseistä muunnelmaa, olen kuitenkin valinnut esimerkkipeliksi seuraavan Aljechinin mestariteoksen, joka löytyy kommentoituna kehuja keränneestä kirjasta "Kaksi shakkineroa" [Konsala, 1991, s.394-395].

Tarrasch – Aljechin

Pistyan, 1922

1.d4 Rf6

2.Rf3 e6

3.c4 c5

4.d5 b5

Blumenfeldin gambiitin avainsiirto. Musta saa sotilaan hinnalla vahvan sotilaskeskustan.

5.dxe6 fxe6

6.cxb5 d5

7.e3 Ld6

8.Rc3 0-0

9.Le2 Lb7

10.b3 Rbd7

11.Lb2 De7

12.0-0 Tad8

13.Dc2 e5

14.Tfe1

Tässä Aljechinin oma kommentti asemasta: "Voidakseen suojata ruudun h2 ratsulla f1:stä. Tästä eteenpäin valkea puolustautuu parhaalla tavalla, mutta hän joutui jo avauksessa toivottomasti alakynteen luovuttaessaan keskustan vastustajalleen."

Katso kuvio 4.

Kuvio 4

Esitettäköön vielä lyhyesti tämän kauneuspalkinnon voittaneen pelin loppu ilman kommentteja:
14. - e4 15.Rd2 Re5 16.Rd1 Rg4 17.Lxg4 Rxg4 18.Rf1 Qg5! 19.h3 Rh6 20.Kh1 Rf5 21.Rh2 d4! 22.Lb1 d3 23.Dc4+ Kh8 24.Lc1 Rg3+! 25.Kg1 Ld5 26.Da4 Re2+ 27.Kh1 Tf7 28.Qa6 h5! 29.b6 Rg3+! 30.Kg1 axb6 31.Dxb6 d2 32.Tf1 Rxf1 33.Rxf1 Le6!! 34.Kh1 Lxh3! 45.gxh3 Tf3 36.Rg3 h4 37.Lf6 Dxf6 38.Rxe4 Txh3+ (0-1)

Musta siis voitti vakuuttavasti, mutta palataan nyt kuvion 4 asemaan siirron 14.Te1 jälkeen ja analysoidaan asema yksinkertaisella keskustakaaviolla:

Valkean keskustan hallinta:

Ta1:	1+1+1+1+2 = 6
Te1:	1+1+1+1+1+1+2 = 8
Kg1:	1+1+2+2+1 = 9
sa2:	2
Lb2:	1+1+1+4 = 7
Dc2:	1+1+1+2+2+4+4+8+4+2+1+2+2 = 34
Le2:	1+1+4+4+2+4 = 16
sf2:	4+2 = 6
sg2:	4+1 = 5
sh2:	2
sb3:	1+4 = 5
Rc3:	1+1+1+2+1+2+8+8 = 24
se3:	8+4 = 12
Rf3:	2+1+1+1+8+8+2+1 = 24
sb5:	1+4 = 5

Yhteensä: 165

Mustan keskustan hallinta:

Te8:	1+1+1+2+1+1 = 7
Tf8:	1+1+1+2+4 = 7
Kg8:	1+1+2+2+1 = 9

sa2:	2
Lb2:	1+1+1+4+8 = 15
Rd7:	1+2+4+8+4+1 = 20
De7:	1+1+1+2+2+2+4+4+4+8 = 29
sg7:	4+1 = 5
sh7:	2
Ld6:	1+2+1+2+4+10 = 20
Rf6:	1+2+1+1+2+1+8+8 = 24
c5:	2+8 = 10
sd5:	4+8 = 12
se5:	8+4 = 12

Yhteensä: 174

Keskustan hallinnan etu on siis valkealla 9 pisteen turvin. Arvioisin, jota myös shakkiohjelmien asema-arviot tukevat, että painokerroin suhteessa materiaaliin olisi noin 0,1. Kun otetaan painokertoimeksi tämä 0,1 niin saadaan kaava:

keskustan hallinta = 0,1 x keskustan hallinta

Ja nyt

keskustan hallinta = 0,1 x (174 – 165) = 0,1 x 9 = 0,9

eli mustan keskustan hallinta korvaa uhratun sotilaan.

3.6.2 Keskustaa reunoillakin suosiva kaavio

Edellisessä kaaviossa ei oteta huomioon sitä, että kaikki reunaruudut eivät ole saman arvoisia. Kulmaruutujen uhkaaminen on hyödyttömämpää kuin reunan keskustaruutujen uhkaaminen.

Kun otetaan huomioon, että on kannattavaa uhata keskustaa reunallakin, Levy ehdottaa seuraavanlaista kaaviota [Levy, 1984, s.19]:

1	2	3	4	4	3	2	1
2	5	6	7	7	6	5	2
3	6	8	9	9	8	6	3
4	7	9	10	10	9	7	4
4	7	9	10	10	9	7	4
3	6	8	9	9	8	6	3
2	5	6	7	7	6	5	2
1	2	3	4	4	3	2	1

Kaavio 4

Tämä kaavio eroaa edellisestä myös siinä, että nyt aivan terävintä keskustaa ei suosita yhtä jyrkästi verrattuna ympäristöönsä. Tämän seurauksena kaavio ei arvosta yhtä paljon sotilaskeskustaa kuin edellinen kaavio.

Katsotaan miten kaavio käsittelee kuvion 4 tilannetta.

Valkean keskustan hallinta:

Ta1:	2+3+4+4+2 = 15
Te1:	1+2+3+4+3+2+7 = 22
Kg1:	3+1+6+5+2 = 17
sa2:	6
Lb2:	1+3+3+8 = 15
Dc2:	2+3+4+5+6+8+9+10+9+6+2+7+7 = 78
Le2:	4+3+9+9+7+8 = 40
sf2:	9+6 = 15
sg2:	8+3 = 11
sh2:	6
sb3:	4+9 = 13
Rc3:	2+2+4+7+10+10+7+4 = 46
se3:	10+9 = 19
Rf3:	2+2+7+4+10+10+7+4 = 46
sb5:	3+8 = 11

Yhteensä: 360

Mustan keskustan hallinta:

Te8:	1+2+3+7+4+3 = 20
Tf8:	4+4+2+6+8 = 24
Kg8:	3+1+6+5+2 = 17
sa2:	6
Lb2:	1+3+3+8+10 = 25
Rd7:	2+6+9+10+8+3 = 38
De7:	4+4+3+7+6+5+9+9+8+10 = 65
sg7:	8+3 = 11
sh7:	6
Ld6:	2+6+3+7+9+10 = 37
Rf6:	4+7+2+2+4+7+10+10 = 46
c5:	7+10 = 17
sd5:	9+10 = 19
se5:	10+9 = 19

Yhteensä: 350

Eli tämän kaavion mukaan valkealla olisi paremmin keskusta hallussa! Shakillisesti tämä ei kuitenkaan voi pitää paikkaansa. Tämä kaavio osaisi luultavasti arvioida oikein asemat, missä jompi kumpi on miehittänyt koko keskustan upseereillaan, mutta ei asemia, missä toisella on vankka sotilaskeskusta ja keskusta täysin hallussa, mutta kummankin painostus keskustaan on suunnilleen yhtä suuri.

3.6.3 Oma yhdistelmäkaavio

Yhdistämällä kaksi edellistä kaaviota toisiinsa, olen kehittänyt seuraavanlaisen kaavion:

2	3	4	5	5	4	3	2
3	6	8	10	10	8	6	3
4	8	18	20	20	18	8	4
5	10	20	40	40	20	10	5
5	10	20	40	40	20	10	5
4	8	18	20	20	18	8	4
3	6	8	10	10	8	6	3
2	3	4	5	5	4	3	2

Kaavio 5

Tässä edetään reunan keskeltä keskustaan täsmälleen samalla jyrkällä suhteella 1:2 kuin kaaviossa 3, mutta myös huomioidaan kaavion 4 ominaisuus, jossa suositaan keskustaa reunallakin.

Katsotaan miten tämä hybridikaavio käsittelee edellistä kuvion 4 asemaa.

Valkean keskustan hallinta:

Ta1:	3+4+5+5+3 = 20
Te1:	2+3+4+5+4+3+10 = 31
Kg1:	4+2+8+6+3 = 23
sa2:	8
Lb2:	2+4+4+18 = 28
Dc2:	3+4+5+6+8+18+20+40+20+8+3+10+10 = 155
Le2:	5+4+20+20+10+18 = 77
sf2:	20+8 = 28
sg2:	18+4 = 22
sh2:	8
sb3:	5+20 = 25
Rc3:	3+3+5+10+40+40+10+5 = 116
se3:	40+20 = 60

Rf3: 3+3+10+5+40+40+10+5 = 116

sb5: 4+18 = 22

Yhteensä: 739

Mustan keskustan hallinta:

Te8: 2+3+4+10+5+4 = 28

Tf8: 5+5+3+8+18 = 39

Kg8: 4+2+8+6+3 = 23

sa2: 8

Lb2: 2+4+4+18+40 = 68

Rd7: 3+8+20+40+18+4 = 93

De7: 5+5+4+10+8+6+20+20+18+40 = 136

sg7: 18+4 = 22

sh7: 8

Ld6: 3+8+4+10+20+40 = 85

Rf6: 5+10+3+3+5+10+40+40 = 116

c5: 10+40 = 50

sd5: 20+40 = 60

se5: 40+20 = 60

Yhteensä: 796

Tämän kaavan mukaan mustalla on keskustan hallinta pistein 796 – 739 = 57. Suhteelliseksi painokertoimeksi saadaan 0,1/5 = 0,02.

Eli nyt
keskustan hallinta = 0,02 x (796 – 739) = 0,02 x 57 = 1,14

Tämän mukaan mustalla on vahvan keskustan ansiosta vähintään riittävä korvaus sotilaasta, mitä tukee myös Aljechinin asema-arvio.

3.6.4 Laudan toiselle puolelle painostava keskustakaavio

Ohjelmasta voidaan tehdä hyökkäävämpi antamalla enemmän pisteitä vastustajan puolella oleville ruuduille. Tämä voidaan tehdä yksinkertaisesti lisäämällä jokin luku, esimerkiksi 1, jokaiseen vastustajan puolella olevan ruudun arvoon. Ohjelmasta voidaan tehdä vieläkin hyökkäävämpi suurentamalla tuntuvasti arvoja, kun edetään kohti vastustajan takariviä. David Levy ehdottaa mallia, jossa edellisen kohdan 3.6.2-malliin lisätään jokaisen ruudun arvoa skaalalla 1-8 sen mukaan millä rivillä ollaan [Levy, 1984, s. 19]. Hän ottaa lähtökohdaksi kaavion 4, jolloin saadaan seuraavanlaiset kaaviot:

Valkean näkökulmasta:

9	10	11	12	12	11	10	9
9	12	13	14	14	13	12	9
9	12	13	14	14	13	12	9
9	12	14	15	15	14	12	9
8	11	13	14	14	13	11	8
6	9	11	12	12	11	9	6
4	7	8	9	9	8	7	4
2	3	4	5	5	4	3	2

Kaavio 6

Mustan näkökulmasta:

2	3	4	5	5	4	3	2
4	7	8	9	9	8	7	4
6	9	11	12	12	11	9	6
8	11	13	14	14	13	11	8
9	12	14	15	15	14	12	9
9	12	13	14	14	13	12	9
9	12	13	14	14	13	12	9
9	10	11	12	12	11	10	9

Kaavio 7

Tätä samaa ideaa voi ilman muuta soveltaa myös mihin tahansa muuhun neutraaliin keskustakaavioon, kuten kaavioon 3 tai kaavioon 5.

Ohjelmasta voidaan vaihtoehtoisesti tehdä passiivisesti pelaava antamalla suurempia arvoja oman laudan puoliskon ruuduista. Esimerkiksi edellinen mustan näkökulmasta hyökkäävä kaavio (kaavio 7) voisi vaihtoehtoisesti olla valkean näkökulmasta puolustustava kaavio.

3.7 Kuninkaan turvallisuus

Kuninkaan turvallisuus on tärkeä asia, joka voidaan mallintaa kuten ihmisetkin tekevät seuraavasti [Levy, 1984, s.23]

a) Kuningas tulisi ympäröidä omilla sotilailla ja upseereilla niin hyvin kuin mahdollista
b) Sotilaat ovat parempia suojanappuloita kuin upseerit
c) Suojatut suojanappulat ovat parempia kuin suojaamattomat suojanappulat
d) Mitä lähempänä kuningasta suojanappula sijaitsee, sitä paremmin se suojaa kuningasta.

Nappulan arvo kuninkaan suojaajana voidaan nyt määritellä esimerkiksi seuraavasti:
läheisyys omaan kuninkaaseen / muokattu nappulan arvo

Kuninkaan viereisillä ruuduilla läheisyys kuninkaaseen on 8 ja niistä ruudun eteenpäin edelleen 4, 2, 1. Tätä kauempana olevat ruudut ovat niin kaukana kuninkaasta, ettei niissä olevasta nappulasta ole hyötyä kuninkaan suojaamisen tarkoituksessa, joten näistä ruuduista ei saa pistettäkään. Muokattu nappulan arvo on todellinen nappulan arvo, jos nappula on suojaamaton tai nappulan arvo / (n+1), jos nappulaa suojaa n omaa nappulaa (mukaan lukien sotilaat ja kuningas).

Kuninkaan kokonaisturvallisuus lasketaan summaamalla yhteen kaikkien oman kuninkaan väristen nappuloiden suojausarvot.

Havainnollistetaan asiaa esimerkillä. Valitaan muunnelmaksi sisilialaisen puolustuksen terävä kaksiteräisen miekan maineessa oleva lohikäärmemuunnelma, jossa pelaajat yleensä linnoittuvat eri puolille ja hyökkäävät kilpaa vastustajan kuningasta vastaan. Tässä asema pelistä Donchev – Semkov:

Kuvio 5

Vaikka musta on onnistunut hankkimaan puoliavoimen b-linjan, valkean kuninkaalla ei ole toistaiseksi mitään pelättävää. Valkealla sen sijaan on vaarallinen hyökkäys meneillään ja musta kuningas kärsii siitä, että sotilasmuuri kuninkaan edessä on edennyt liiaksi. Valkealla pitäisi siis olla etua, mutta katsotaan mitä kaava sanoo.

Valkean kuninkaan turvallisuus:

Td1:	8 / (5/3) = 4,8
Th1:	0 / (5/2) = 0
sa2:	4 / (1/1) = 4,0
sb2:	8 / (1/2) = 16,0
sc2:	8 / (1/3) = 24,0
De3:	4 / (9/3) = 1,3
Rd4:	2 / (3/3) = 2,0
se4:	2 / (1/2) = 4,0
Rd5:	1 / (3/2) = 0,7
sf5:	1 / (1/3) = 3,0
Lh6:	0 / (3/2) = 0

Yhteensä: 59,8

Mustan kuninkaan turvallisuus:

Dd8:	2 / (9/2) = 0,4
Te8:	4 / (5/3) = 2,4
Ld7:	2 / (3/2) = 1,3
Lg7:	8 / (3/3) = 8,0
sd6:	2 / (1/1) = 2,0
se6:	4 / (1/3) = 12,0
sf6:	4 / (1/4) = 16,0
sg6:	4 / (1/1) = 4,0
sa5:	0 / (1/3) = 0
Tc5:	1 / (5/2) = 0,4
Rh5:	2 / (3/2) = 1,3
sc4:	1 / (1/2) = 2,0
sg4:	1 / (1/1) = 1,0

Yhteensä: 50,8

Valkean etu kuninkaan turvallisuudessa on siis 9 yksikköä.

Peli jatkui 1.fxg6 exd5 2.Lxg7 Kxg7 3.Txh5 (0-1).
Musta ei voi estää valkean mattihyökkäystä h-linjalta, joten musta luovutti.

Otetaan toiseksi esimerkiksi asema omasta pelistäni. Suluissa olevat pelivahvuutta kuvaavat vahvuusluvut ovat pikapelivahvuuslukuja ja ne ovat juuri ennen turnausta voimassa olleet, jonka jälkeen kummankin luku nousi.

Aki Seppälä(2003) – Teemu Pudas(2105)
Shakin joukkuepikapelin SM-kilpailut, 2004

Katso kuvio 6.

Kuvio 6

Valkean kuninkaan turvallisuus:

sc1 1 / 1 = 1,0

sh2: 8 / (1/2) = 16,0

sg3: 4/ (1/2) = 8

Yhteensä: 25

Mustan kuninkaan turvallisuus:

sc7: 1 / 1 = 1,0

sh7: 8 / (1/2) = 16,0

Rc6: 3 / (3/3) = 1,0

Dc5: 1 / 9 = 0,1

sc5: 1 / (1/2) = 2,0

sd5 1 / (½) = 2,0

Yhteensä: 22,1

Kaavan mukaan valkean etu kuninkaan turvallisuudessa olisi vain 2,9 yksikköä. Musta sai suojatuista sotilaistaan c5 ja d5 yhteensä neljä yksikköä kuninkaan turvallisuuteen, mikä tuntuu suurelta liioittelulta ainakin tässä tilanteessa. Sotilaista ei ole mitään hyötyä mustan kuninkaalle,

sillä ne ovat aivan liian kaukana. Sama ilmiö on huomattavissa myös edellisessä asemassa, jossa valkean todella kaukana kuninkaasta oleva suojattu sotilas f5:ssa antaa lähes yhtä paljon pisteitä kuin tärkeä kuningasasemaa suojaava sotilas a2:ssa. Mielestäni uloimman vyöhykkeen voisi hyvin vaikka poistaakin, sillä näissä olevat nappulat ovat jo niin kaukana omasta kuninkaasta, ettei niillä useinkaan ole mitään merkitystä oman kuninkaan suojaamisessa. Jos tämä uloin vyöhyke poistettaisiin, valkean etu kuninkaan turvallisuudessa olisi nyt selvä 8 yksikköä, joka kuvaa tilannetta paremmin. Toisaalta tässä voisi myös pohtia erästä asiaa, jota luultavasti mikään shakkiohjelma ei osaa ottaa huomioon. Nimittäin valkean sotilas g7:ssa itse asiassa suojaa hieman mustan kuningasta. Musta jättikin tahallaan heti avauksessa sotilaan syömättä ilmeisenä tarkoituksenaan jättää se oman kuninkaan kilveksi. Idea toimikin hyvin aina tähän asti ja tälläkin hetkellä sotilas estää valkeaa tekemästä yhden siirron matin Dg7X. Valkealla on tässä kuitenkin taktinen kikka, joka perustuu siihen, että sotilas g7:ssa myös rajoittaa mustan kuninkaan liikkumista, joten vastustajan sotilaan ottaminen oman kuninkaan kilveksi sisältää myös riskejä, eikä ole helppoa sanoa tai määritellä milloin se kannattaa ja milloin ei.

Valkea jatkoi 1.Txc6!! ja musta luovutti välittömästi (1-0). Vaihtoehdot olivat mustalle lohduttomia: 1. - bxc6 2.De6X tai 1. - Dxc6 2.Df8+! Txf8 3.gxf8DX.

Esimerkit osoittavat, että kuninkaan turvallisuus on tärkeä asia. Tätä arvotusfunktion osa-aluetta voidaan edelleen laajentaa. Voidaan esimerkiksi alarivin mattia vastaan antaa bonusta kuninkaan pakoruuduista, jotka useimmiten syntyvät siirroilla h2 tai h7. Itse en kuitenkaan suosittelisi näitä siirtoja tehtäväksi ainakaan kovin aikaisin pelissä. Nämä siirrot heikentävät sotilasasemaa kuninkaan edessä. Varsinkin jos pelaajat ovat linnoittuneet eri puolille, tämä voi olla kohtalokasta. Sotilashyökkäys on tehokkain kuningashyökkäyksen laji, jos kuninkaat ovat linnoittuneet eri puolille ja on selvää, että vastustajan harkitsemattomat sotilaan siirrot mahdollistavat omien sotilaiden avata vastustajan kuningasasema nopeammin verrattuna siihen, että hänen sotilaansa olisivat alkuperäisillä paikoillaan. Olen omissakin peleissä hyödyntänyt tällaista vastustajan tekemää strategista virhettä.

3.8 Nappuloihin kohdistuvat uhkaukset ja niiden suojaukset

Edelliset alaluvut ovat käsitelleet strategisia asioita. Näiden avulla ohjelma saadaan "ymmärtämään" asema ja tuottamaan asema-arvio. Lisäksi on otettava huomioon taktiset

mahdollisuudet, koska materiaalin voitto tai matti voi olla pitkän pakottavan siirtosarjan päässä. Tässä alaluvussa tutkitaan, miten ohjelma saadaan arvioimaan taktisten mahdollisuuksien suuruus.

Jos muut asiat ovat tasan, se pelaaja, jonka nappulat ovat paremmin suojatut ja/tai jonka vastustajan nappulat ovat suuremman hyökkäyksen kohteena, omaa etua. Vaikka suoraa voittoa ei olisikaan, niin taktinen potentiaali on olemassa.

Otetaan esimerkiksi ranskalaisen puolustuksen Winawerin muunnelma, joka etenee seuraavasti: 1.e4 e6 2.d4 d5 3.Rc3 Lb4

Katso kuvio 7.

Kuvio 7

Tässä asemassa valkealla on vain yksi hyökkäys: exd5. Mustalla sen sijaan on kaksi hyökkäystä: dxe4 ja b4-lähetti uhkaa c3-ratsua. Joten tällä hetkellä mustalla on suurempi hyökkäyspotentiaali.

Hyökkäyspotentiaali voitaisiin määritellä vastustajalta nappulan syövien laillisten siirtojen lukumäärällä. David Levy antaa kuitenkin tarkemman määrityksen [Levy, 1984, s.28]:

(ATTACKEDval/ATTACKINGval) / suojaajien lukumäärä + 0.25)

Edellisen aseman hyökkäyspotentiaalit laskettaisin seuraavasti:

Sotilas e4 sotilaaseen d5: (1/1) / 2,25 = 0,4
Sotilas d5 sotilaaseen e4: (1/1) / 0,25 = 4,0
Lähetti b4 ratsuun c3: (3/3) / 1,25 = 0,8

Mustalla on siis 4,4 yksikköä parempi hyökkäyspotentiaali.

Napppuloiden suojauksen suuruuden määrittelemiseen tarvitsee ottaa huomioon suojaavien nappuloiden lukumäärä ja suojaavien nappuloiden arvo. Tämä voidaan tehdä yksinkertaisesti summaamalla yhteen kuhunkin nappulaan kohdistuvat suojaukset kaavalla
nappuloiden lukumäärä / nappulan arvo.
Tämä aiheuttaa sen, että on edullista suojata omat nappulat mahdollisimman vähäarvoisilla omilla nappuloilla. Yleisenä periaatteenahan puolustuksessa on, että yritetään suojata omat nappulat sotilailla, jos mahdollista, muutoin ratsuilla ja läheteillä, jos mahdollista, jne.. Jos käyttää kaikkein arvokkaimpia nappuloita puolustamiseen, ne eivät ole niin helposti saatavissa hyökkäystarpeisiin.

Edellisessä asemassa valkean puolustus olisi:

Lc1: 1 / 5 + 1 / 9 = 0,3
Dd1: 1 / 4 = 0,3
Lf1: 1 / 4 = 0,3
Rg1: 1 / 5 = 0,2
sa2: 1 / 5 = 0,2
sb2: 1 / 3 = 0,3
sc2: 1 / 9 = 0,1
sf2: 1 / 4 = 0,3
sg2: 1 / 3 = 0,3

Tässä on huomattava, että kuninkaan arvo hyökkäävänä ja puolustavana nappulana on arvioitu neljän sotilaan arvoiseksi.

Mustan puolustuksen arvo olisi:

Rb8: 1 / 5 = 0,2
Lc8: 1 / 9 = 0,1

Dd8: 1 / 4 = 0,3

Rg8: 1 / 9 = 0,1

sa7: 1 / 5 = 0,2

sb7: 1 / 3 = 0,3

sc7: 1 / 9 = 0,1

sf7: 1 / 4 = 0,3

sh7: 1 / 5 = 0,2

se6: 1 / 3 + 1 / 1 = 1,3

sd5: 1 / 9 + 1 / 1 = 1,1

Hyökkäyspotentiaalin kaavaa voidaan nyt parantaa ottamalla huomioon myös suojaavien nappuloiden arvot:

(ATTACKEDval / ATTACKINGval) / DEFENCE

Nappulaan kohdistuva kokonaishyökkäys lasketaan kaavalla:

Piece Attack (PA) = (ATTACKEDval x summa (1 / ATTACKINGval)) / summa (1 / DEFENDINGval)

Otetaan esimerkiksi hyökkäyspelin mestarin Mihail Talin peli Gurgenidze – Tal vuodelta 1957. Katso kuvio 8.

Kuvio 8

Seuraavassa taulukossa huomioidaan vain ne nappulat, joita uhataan.

Nappula	ATTACKEDval	1/ATTACKINGval	1/DEFENDINGval	PA
vsb2	1	1 / 3	1 / 3 + 1 / 9	0,75
vsg2	1	1 / 9	1 / 3 + 1 / 3 + 1 / 4	0,12
vRe3	3	1 / 3 + 1 / 3 + 1/5+1/5	1 / 3 + 1 / 3 + 1 / 5	1,63
vRc4	3	1 / 3	1 / 3 + 1 / 9	2,25
mLd5	3	1 / 3 + 1 / 3	1 / 5	10,00
mpg6	1	1 / 9	1	0,11

Taulukko 1

PA tarkoittaa siis kyseiseen nappulaan kohdistuvaa uhkaa ja lasketaan esim. vsb2:n tapauksessa (1 x (1 / 3)) / (1/3 + 1/9) = 0,75. Lisäksi on huomattava, että myös röntgenhyökkäykset lasketaan eli myös mTe8 hyökkää vRe3:aa vastaan. Huomionarvoista, mutta itsestään selvää, on myös, että vRc4 ei hyökkää mitään nappulaa vastaan, koska on kiinnitetty eikä voi liikkua.

Analysoidaan nyt hieman taulukon tuloksia. On selvää, että valkealla on vakava hyökkäys mustan ruudussa d5 olevaa ratsua vastaan, kuten suuri PA-arvo 10 osoittaa. On myös helppo nähdä, että tämä on valkean ainut merkittävä uhka mustan nappuloita vastaan. Valkean toinen uhka eli kuningattaren hyökkäys g6-sotilasta vastaan on lähes merkityksetön, kuten pieni PA-arvo 0,11 kuvaa. Mustalla sen sijaan on hyökkäysmahdollisuuksia peräti neljään eri ruutuun, mikä osoittaa, että mustalla on kovaa painetta valkean asemaa vastaan.

Koska kummallakin on merkittäviä PA-arvoja, asema on hyvin terävä. Hyvä ohjelma huomioisi tässä, että täten on kannattavaa ensin laskea syvälle ja sitten vasta arvioida asemia arvotusfuktiolla. Älykäs ohjelma voisi lisäksi alentaa valkean hyökkäyksen arvoa ruutuun d5 seuraavalla tavalla: jos valkea lyö ratsun ratsulla, hän joutuu heti mattiin (1.Rxd5? Dg1X) ja jatko 1.Lxd5 Txd5 2.Rxd5 Dg1X on yhtä huono valkealle, joten on helppo nähdä, että valkealla ei ole todellista uhkaa voittaa materiaalia d5:ssa. Valkealla ei ole hyökkäysmahdollisuuksia mustan kuningasta vastaan, mutta mustan painostus valkean asemaan on todellista.

3.9 Loppupelipiirteet

Loppupelissä tulevat eri asiat tärkeiksi kuin alkupelissä ja keskipelissä. Tärkeimpiä asioita loppupelissä ovat vapaasotilaan muodostaminen ja sen eteneminen ja ilman sotilaita olevassa loppupelissä matin pakottaminen esimerkiksi kuninkaalla ja tornilla pelkkää kuningasta vastaan.

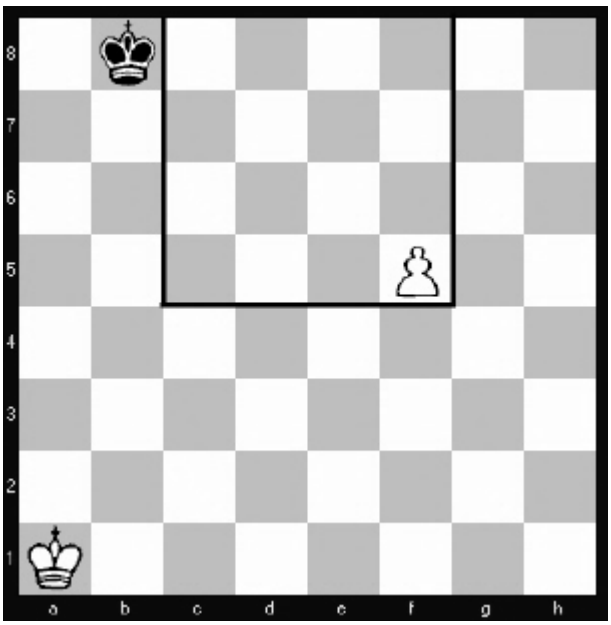
Loppupeli on koneelle kaikista vaikein osa-alue, koska voittosuunnitelma voi sisältää jopa 20 siirtoa. Ihminen sen sijaan tietää oikeat voittosuunnitelmat, eikä hänen tarvitse laskea kaikkea loppuun niinkuin koneen.

3.9.1 Yleinen loppupeli ja sotilasloppupeli

Yleisessä loppupelissä on jäljellä sotilaita ja upseereita. Sotilasloppupelissä on jäljellä vain sotilaita. Molemmissa tapauksissa tavoite on sama: vapaasotilaan muodostaminen, sen eteneminen ja lopulta korottaminen kuningattareksi.

Luvussa 3.5 todettiin jo kuinka arvokkaita vapaasotilaat ovat. Seuraavat asiat puolestaan auttavat vapaasotilasta etenemään:
i) Annetaan bonusta tornista vapaasotilaan takana.
ii) Rangaistaan jokaisesta omasta nappulasta, joka tukkii oman vapaasotilaan reitin. (Vastaavasti vastustajan vapaasotilasta vastaan puolustaessa voidaan antaa bonusta, jos oma nappula tukkii vastustajan vapaasotilaan reitin.)
iii) Tutkitaan onko vastustajan kuningas ulkona oman vapaasotilaan neliöstä. Sotilasloppupelissä vain kuningas voi estää vapaasotilasta korottumasta, joten on tärkeää tietää saavuttaako vastustajan kuningas vapaasotilaan vai ei, jos sotilas lähtee etenemään.

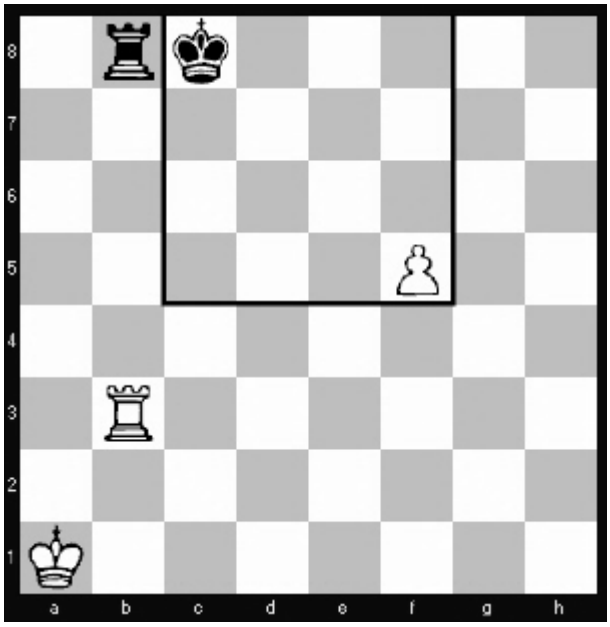
Otetaan esimerkiksi seuraava asema:



Kuvio 9

Tässä vapaasotilaan neliö käsittää alueen f5-f8-c8-c5. Jos olisi mustan siirtovuoro, mustan kuningas voisi siirtyä tähän neliöön ja tällöin tuloksena olisi tasapeli, sillä kuningas saisi sotilaan kiinni, esimerkiksi: 1. - Kc8 2.f6 Kd8 3.f7 Ke7 4.f8D+ Kxd8 ½-½.
Jos taas olisi valkean vuoro, musta kuningas ei saa sotilasta kiinni: 1.f6 Kc8 2.f7 Kd8 3.f8D+ ja valkea voittaa. Testasin asemaa Rebel 10-ohjelmalla, joka löysi valkean voiton jo yhden puolisiirron syvyydellä. Ohjelmaan oli siis koodattu neliösääntö.

Neliösäännön tuntemisesta on hyötyä myös yleisessä loppupelissä. Katso kuvio 10.

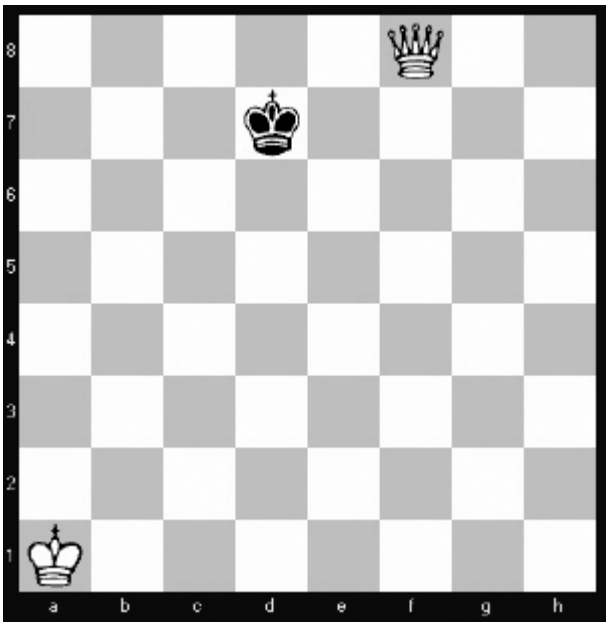


Kuvio 10

Kun ohjelma, kuten Rebel 10, tuntee neliösäännön, se tietää heti, että siirtojen 1.Txb8+ Kxb8 jälkeen mustan kuningas on ulkona sotilaan neliöstä ja valkea voittaa, eikä sen sen tarvitse laskea jatkoa 2.f5 Kc8 3.f7 4.f8D+.

3.9.2 Matin pakottaminen upseereilla

Kun sotilaita ei enää ole, tavoitteeksi vaihtuu matin tekeminen upseereilla. Otetaan esimerkiksi edellisen alaluvun tilanne, jossa sotilas olisi päässyt korottumaan kuningattareksi ja nyt valkean pitäisi tehdä matti. Katso kuvio 11.



Kuvio 11

Testasin asemaa Rebel 10 - ohjelmalla ja yhden puolisiirron syvyydelle menevässä pelimoodissa se ei osannut tehdä mattia, vaan peli päättyi tasan 50 siirron kuluttua. Ohjelma oli ilmeisesti asetettu suosimaan omaa kuningasta keskustassa kuten loppupelissä usein tehdään, sillä kuningas pyöri koko ajan keskustassa. Tosin jo kahden puolisiirron syvyydelle menevässä pelimoodissa ohjelma näki matin.

Ohjelma saadaan tekemään matti upseereilla ottamalla mukaan hyvä kuninkaan kulmaan ajava algoritmi. Ohjelma täytyy saada havaitsemaan miten parhaiten ajaa kuningas kulmaan (tai ratsun ja lähetin tapauksessa, oikean värisen lähetin kulmaan). Tämä voidaan käytännössä tehdä niin, että annetaan vastustajan kuninkaalle hyviä arvoja keskustassa ja sitä huonompia mitä lähemmäksi

kulmaa se joutuu. Lisäksi ohjelman pitäisi tietää, että oman kuninkaan olo vastustajan kuninkaan lähellä on tärkeä asia.

Gillogly esittää seuraavanlaisen käytännön ratkaisun [Gillogly, 1972]:

Tehdään pieni kahden puolisiirron haku ja vaihdetaan arvotusfunktioksi funktio, joka sisältää seuraavat 4 kohtaa:

1) -32 x vastustajan kuninkaan sijainti keskustakaaviossa (Kaavio 4': muuten samanlainen kuin kaavio 4, mutta reunaruutujen arvoa on vähennetty yhdellä ja muiden ruutujen arvoa kahdella)
2) 2 x vastustajan kuninkaan sijainti ohjelman kuninkaan kuningaskaaviossa (Kaavio 8)
3) ohjelman kuninkaan sijainti keskustakaaviossa (Kaavio 4')
4) ohjelman nappuloiden sijainnista ohjelman kuninkaan kuningaskaaviossa (kaavio 8) saatavat pisteet jaettuna ohjelman nappuloiden lukumäärällä (näin saadaan nappulat pysymään kuninkaan lähellä)

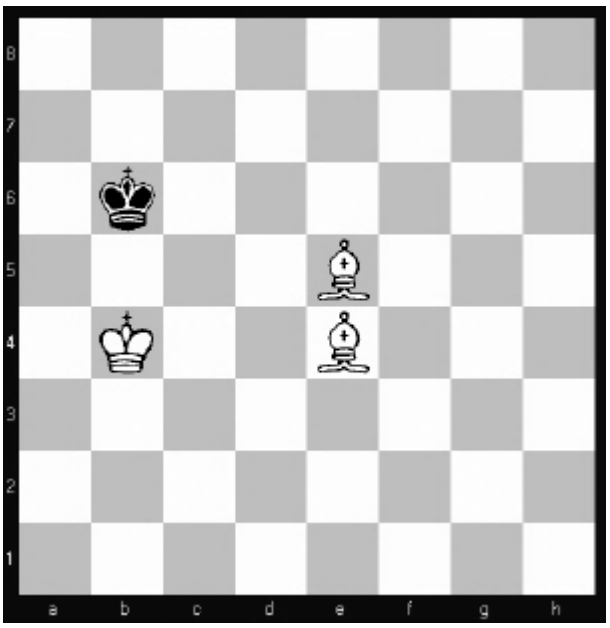
Kuningaskaavio on seuraavanlainen:

4	4	5	6	5	4	4
4	8	10	9	10	8	4
5	10	10	10	10	10	5
6	9	10	K	10	9	6
5	10	10	10	10	10	5
4	8	10	10	10	10	5
4	8	10	9	10	8	4
4	4	5	6	5	4	4

Kaavio 8

Eräs seikka, jota ei usein huomioida, on se, että on parempi saavuttaa jokin tavoite esimerkiksi kolmessa puolisiirrossa kuin viidessä puolisiirrossa.

Otetaan esimerkiksi seuraava asema:



Kuvio 12

Oletetaan, että ohjelma puntaroisi tässä kahden vaihtoehdon välillä:

MUUNNELMA A: 1.Ld5 Ka6 2.Ld6

MUUNNELMA B: 1.Lf4 Ka7 2.Ld5 Ka6 3.Ld6

Molemmat muunnelmat johtavat täsmälleen samaan asemaan. Oletetaan, että ohjelma valitsee tällaisissa tilanteissa satunnaisesti ja valitsee nyt muunnelman B: 1.Lf4 Ka7. Ohjelma näkee nyt muunnelman 2.Ld5 Ka6 3.Ld6, mutta se näkee myös muunnelman 2.Lg3 Kb6 3.Ld5 Ka6 4.Ld6, joka johtaa täsmälleen samaan asemaan. Ohjelma saattaisi nyt valita jälkimmäisen muunnelman jne., jolloin edistyminen on hidasta. Ongelma voidaan korjata antamalla johdossa olevalle puolelle pieni rangaistus jokaisesta aseman arvioinnista siten, että se on sitä suurempi mitä syvemmällä hakupuussa ollaan. Häviöllä oleva puoli puolestaan saisi bonusta mitä syvemmällä ollaan. Tuloksena ohjelma yrittäisi voittaa tai parantaa asemaansa niin nopeasti kuin mahdollista ja vastaavasti häviöllä ollessa pitkittää häviötä tai aseman huononemista niin pitkään kuin mahdollista.

3.10 Muita osa-alueita

Mahdollisia eri osa-alueita on lukemattomia. Esimerkiksi Deep Blue:n arvotusfunktio sisältää yli 8000 eri osa-aluetta [Campbell, 2002].

Seuraavassa on lyhyt luettelo joistakin vielä mainitsemattomista asioista, joita voisi olla hyödyllistä

ottaa huomioon arvotusfunktiossa.

- Torni 7. rivillä: annetaan bonusta, jos valkea saa tornin 7. riville tai vastaavasti, jos musta saa tornin 2.riville.

- Torni avoimelle linjalle: annetaan bonusta, jos torni valtaa avoimen linjan.

- Kiinnitys: annetaan bonusta, jos vastustajan nappula on kiinnitetty ja jos kiinnitys on pysyvä, bonusta annetaan vieläkin enemmän.

- Lähettipari: lähettiparin omaavalle annetaan bonusta.

- Eri väriset lähetit: eri väriset lähetit alentavat asema-arviota, koska eri värisillä läheteillä on vaikea voittaa.

- Kaukana toisistaan olevat vapaasotilaat: jos kuitenkin joudutaan eriväristen lähettien loppupeliin, kaukana toisistaan olevat vapaasotilaat antavat voiton toiveita.

- Nappuloiden yhteensopivuus: esimerkiksi kuningatar ja ratsu toimivat paremmin yhdessä kuin kuningatar ja lähetti.

- Ansat: taulukko tyypillisistä tilanteista, joissa upseeri jää ansaan ja näistä annetaan rangaistuksia.

- Siirtovuoro: annetaan pieni bonus sille pelaajalla, joka on siirtovuorossa.

3.11 Osa-alueiden valinta ja painokertoimien määritys

Tässä luvussa 3 on käsitelty mahdollisia eri osa-alueita mitä arvotusfunktioon voi sisällyttää. Materiaalin huomioiminen (luku 3.1) on välttämätöntä. Kun tähän lisätään vielä toiseksi tärkein asia liikkuvuus (luku 3.2), saadaan jo hyvä lähtökohta arvotusfunktiolle. Ohjelmoija voi tämän jälkeen lisätä osa-alueita oman makunsa mukaan kuin kokki aineksia keitokseen. Hitaissa ympäristöissä tai jos halutaan keskittää laskentatehoa muunnelmien laskemiseen voi olla edullista valita yksinkertainen arvotusfunktio, jossa on vähän osa-alueita ja tällöin aseman arviointi on nopeaa.

Voidaan myös tehdä niin, ettei arvotusfunktio ole koko ajan sama, vaan se muuttuu pelin edetessä. Seuraavassa on karkea hahmotelmaesimerkki kokonaisuudesta. Voidaan määritellä avausvaihe, joka kestää ensimmäiset 8 siirtoa [Gillogly, 1972]. Tässä vaiheessa mukana voisi olla edellisten kahden perusaineksen lisäksi avausvaiheen tärkeät asiat; kehitys (luku 3.3) ja keskustan hallinta(luku 3.6). Tämän jälkeen alkaa keskipeli, jossa voidaan pyrkiä kuningashyökkäykseen ottamalla mukaan hyökkäävä kuningashyökkäyskaavio (luku 3.4.2). Vaihtoehtoisena strategiana keskipelissä voisi myös olla solidi passiivinen asemapeli. Tällöin sotilasaseman pysyminen terveenä on tärkeää (luku 3.5) ja keskustakaavioksi voisi vaihtaa puolustavan keskustakaavion (luku 3.6.4). Loppupelissä pyritään ensin vapaasotilaan muodostamiseen (luku 3.9.1) ja tätä voidaan myös edistää antamalla vapaasotilaasta suurempia arvoja kuin keskipelissä. Kun sotilaita ei enää ole, viimeinen tehtävä on tehdä matti jäljelle jääneillä upseereilla ja arvotusfunktioksi voidaan vaihtaa vastustajan kuningasta kulmassa suosiva arvotusfunktio. (luku 3.9.2).

Kun on valittu, mitä osa-alueita halutaan ottaa mukaan arvotusfunktioon, tehtäväksi jää osa-alueiden painokertoimien määrittäminen.

Arvotusfunktio muodostuu seuraavasti [Hyvönen ym, 1993, s.179]:

aseman arvo = kerroin_1 x tekijä_1 + kerroin_2 x tekijä_2 + ... + kerroin_n x tekijä_n, missä

tekijät ovat arvotusfunktion eri osa-alueita ja
kertoimet ovat näiden osa-alueiden painokertoimia

Tutkitaan luvun lopuksi millaisia eri tapoja on olemassa painokertoimien määrittämiseksi.

3.11.1 Arvaaminen

Suosituin tapa painokertoimien määrittämiseen on tehdä hyvä arvaus ja parantaa tätä arvausta käytännön kokemusten perusteella. Esimerkiksi jos ohjelma ei näytä välittävän tarpeeksi oman kuninkaansa turvallisuudesta (luku 3.7), niin lisätään tämän osa-alueen painoarvoa. Tällä tavalla ovat toimineet useimmat shakkiohjelmoijat.

3.11.2 Systemaattinen testaus

Kasataan arvotusfunktio askel askeleelta. Aloitetaan kahdella osa-alueella ja loogisinta on aloittaa materiaalilla ja liikkuvuudella, koska ne ovat kaksi tärkeintä osa-aluetta. Kaikki muut osa-alueet voidaan väliaikaisesti irrottaa arvotusfunktiosta. Asetetaan materiaalin painokertoimeksi 1. Liikkuvuus asetetaan siten, että kun ohjelma pelaa valkeilla liikkuvuuden painokerroin on 0,1, mutta kun ohjelma pelaa mustilla painokerroin on 0,03. Laitetaan nyt ohjelma pelaamaan sarja pelejä itseään vastaan ja tulos on luultavasti, että musta tekee positiivisen tuloksen. Koska 0,03 on osoittautunut paremmaksi painoarvoksi kuin 0,1, lasketaan painokerrointa suunnilleen kymmenellä prosentilla. Valkean painokerroin on nyt 0,9, mutta mustan painokerroin on edelleen muuttumaton 0,03. Laitetaan ohjelma jälleen pelaamaan sarja pelejä itseään vastaan. Korjataan jälleen häviävän puolen painokerrointa kymmenyksellä. Tätä jatketaan, kunnes pelisarja päättyy suunnilleen tasan. Lopulliseksi painokertoimeksi tulee viimeisen kokeilun kahden version painokertoimien keskiarvo.

Seuraavaksi lisätään uusi osa-alue arvotusfunktioon. Arvataan osa-alueen alaraja ja yläraja. Käytetään näitä rajoja kahden version aloitusarvoina. Annetaan ohjelman taas pelata itseään vastaan ja tarkennetaan näitä rajoja kuten edellä kuvattiin.

Prosessi jatkuu, kunnes kaikki halutut osa-alueet on lisätty ja painokertoimet määritetty.

3.11.3 Automaattinen määritys

Arthur Samuel on kehittänyt täysin automaattisen tavan painokertoimien määrittämiseen [Samuel, 1959]. Hänen alkuperäinen kohteensa oli tosin tammipeli, mutta samaa ideaa voidaan soveltaa myös shakkiin.

Samuelin lähtökohta oli, että jos arvotusfunktio on täydellinen, eteenpäin laskemisen tuloksena saatava asema antaa saman asema-arvion kuin alkuperäinen juuriasema. Nämä aseman arvot ovat itse asiassa usein erilaisia ja tätä eroa hyödynnetään seuraavasti.

Kun suoritetaan haku syvyydelle d (puolisiirroissa), niin tämän aseman arvo on Sback ja Sprev on aseman arvo syvyydeltä d-2. Sback on siis luotettavampi aseman arvo. Näiden erotusta Sback – Sprev kutsutaan deltaksi. Jos delta on positiivinen, niin Sprev on virheellinen ja sitä täytyy korjata siten että positiivisten osa-alueiden painokertoimia suurennetaan ja negatiivisten osa-alueiden

painokertoimia pienennetään. Kääntäen, jos delta on negatiivinen, niin negatiivisten osa-alueiden painokertoimia suurennetaan ja positiivisten osa-alueiden painokertoimia pienennetään.

Sutton loi Samuelin oivallusten pohjalta TD-algoritmin vuonna 1988, mutta sen soveltaminen shakkiin oli pitkään vaikeaa [Furnkrazt, 2002]. Läpimurto tapahtui vasta vuonna 1998, jolloin Baxter, Tridgell ja Weaver loivat algoritmin pohjalta KnighCap-ohjelman, joka kykeni nostamaan suorituslukuaan 1650:sta 2150:aan pelaamalla Internetissä vain 300 peliä kolmen päivän ajan. Yksityiskohtaista tietoa Knight Cap-ohjelmasta ja sen algoritmeista on saatavissa ohjelman tekijöiden artikkelista "Learning to play chess using temporal differences" [Baxter ym, 2000].

4 HAKUTEKNIIKAT

Edellisissä luvuissa esiteltiin miten asemasta tuotetaan lailliset siirrot ja miten asemia arvioidaan. Siis jo näillä menetelmillä kone pelaisi shakkia joten kuten. Shakkiohjelmien menestys perustuu kuitenkin enemmän niiden suorittamaan hakuun. Tässä luvussa tutkitaan hakutekniikoita.

4.1 Esihaku taulukoista

Ennenkuin ohjelma alkaa analysoida asemaa, se tarkistaa ensin löytyykö asema jo jostain sen taulukosta. Jos löytyy, ohjelma voi tehdä suoraan taulukon suositteleman siirron, jolloin ohjelma säästää miettimisaikaa.

4.1.1 Avauskirjasto

Avaussiirroista koostuvaa taulukkoa kutsutaan avauskirjastoksi (opening book). Avauskirjasto ei ole täysin välttämätön, mutta käytännössä jokaisella shakkiohjelmalla on avauskirjasto. Niistä on paljon hyötyä: miettimisaikaa säästyy, pelien avauksista tulee monipuolisempia ja konetta voidaan estää lähtemästä sille sopimattomiin muunnelmiin.

Muutokset ohjelman avauskirjastossa tekee yleensä ihminen. Jos ohjelma pelaa ottelua ihmistä (tai toista ohjelmaa) vastaan ja häviää pelin huonon avauksen takia, ohjelmoijat yleensä muuttavat ohjelman avauskirjastoa siten, ettei vastustaja enää seuraavassa ottelun pelissä pääse samaan muunnelmaan. Yleensä avauskirjaston muutoksista on hyötyä ohjelmalle, mutta joskus niistä voi olla haittaakin. Hyvä esimerkki tästä on ottelu Garry Kasparov – X3D Fritz vuodelta 2003 [Innala, 2004]. Ottelun ensimmäisessä pelissä ohjelma teki avauskirjastostaan hyvän sinne lisätyn siirron kriittisessä asemassa, jossa ohjelmat ovat ennen jatkaneet heikommin. Siirron oli lisännyt ohjelman avauskirjastosta vastannut Alex Kure. Ohjelma saavutti näin tasapelin mustilla, joka oli hyvä tulos. Korkeimmalla tasolla lähtökohta on yleensä, että valkeilla yritetään voittaa ja mustilla pitää tasapeli. Ottelun kolmannessa pelissä Fritzin avaustiimi kuitenkin epäonnistui pahoin. Kasparov oli valmis pelaamaan 1. pelin avausta uudelleen, mutta ohjelma poikkesi tästä avauskirjastonsa pakottamana ja joutui suljettuun asemaan. Suljetut asemat ovat ihmisen ja koneen välisissä peleissä ihmiselle

huomattavasti edullisempia kuin avoimet asemat. Suljetuissa asemissa koneen laskuvoima ei pääse oikeuksiinsa, kun sitä vastoin ihminen osaa tehdä kauaskantoisia strategisia suunnitelmia. Kasparov voittikin helposti pelin tällä antikonestrategialla.

Tutkitaan seuraavaksi kuinka avauskirjasto käytännössä toimii ja miten sitä voi muokata. Esimerkkinä on Colossus Chess X:n avauskirjasto [Bryant, 1989]. Jokainen avauskirjastossa oleva siirto merkitään lipulla, jolla määritetään kuinka hyvä siirto on. Huutomerkki(”!”) tarkoittaa vahvaa siirtoa, tyhjä väli (” ”) normaalia siirtoa, dollarimerkki(”$”) epäilyttävää siirtoa ja kysymysmerkki(”?”) huonoa siirtoa. Kone pyrkii tietenkin tekemään mahdollisimman korkean statuksen omaavia siirtoja ja jos samanarvoisia siirtoja on useita, siirto valitaan satunnaisesti. Ohjelma ei tee vapaaehtoisesti epäilyttäviä eikä huonoja siirtoja. Epäilyttävä siirto on tässä yhteydessä sellainen, joka sinällään on pelikelpoinen, mutta ei sovi ohjelman pelityyliin. Esimerkiksi siirtojen 1.e4! e5! jälkeen Colossus Chess X:n avauskirjasto arvioi siirron 2.f4 epäilyttäväksi. Syy on ilmeinen. Terävä kuningasgambiitti ei sovi ohjelman passiiviseen pelityyliin. Epäilyttävät ja huonot siirrot on kuitenkin hyvä mainita avauskirjastossa ei pelkästään sen takia, että kone osaa itse välttää niitä, vaan myös sen takia, että jos sen vastustaja tekee epätarkan siirron niin kone saa suoraan avauskirjastostaan kumouksen siirrolle. Colossus Chess X:ssä tavallisella käyttäjällä on myös mahdollisuus muokata avauskirjastoa mielensä mukaan. Hän voi esimerkiksi lippuja muuttamalla opettaa koneelle mitkä siirrot ovat hyviä ja mitkä huonoja, lisätä mielenkiintoisia muunnelmia tai pakottaa ohjelman lähtemään tiettyyn muunnelmaan, jota pelaaja haluaa harjoitella.

Avauskirjasto-oppimista on yritetty tehdä automaattiseksi. Ensimmäinen yritys oli juuri edellä mainittu ohjelma Colossus Chess X, johon oli ensimmäisenä kaupallisena ohjelmana lisätty tällainen ominaisuus. Ohjelma lisäsi automaattisesti pelattuja jatkoja erilliseen oppimisavauskirjastoon. Kun ohjelma joutui seuraavan kerran avauksessa asemaan missä se oli jo ollut, se säästi aikaa kun siirron sai otettua suoraan avauskirjastosta. Valitettavasti tästä menetelmästä voi olla myös vakavaa haittaa ohjelmalle. Jos ohjelma menee oppimaan huonon jatkon, sillä ei ole mitään tapaa enää itse myöhemmin korjata virhettään, vaan se toistaa automaattisesti häviösiirron suoraan avauskirjastostaan kerta toisensa jälkeen. Hyvä esimerkki tästä on Wieniläisen avauksen muunnelma 1.e4 e5 2.Rc3 Rf6 3.Lc4 Rc6 4.d3 Lc5 5.f4. Myynnissä olevaan ohjelman PC-versioon oli harmillisesti lipsahtanut ohjelman oppima lisäjatko 5. - d6 6.Ld2 0-0 7.f5. Nykyisillä konetehoilla ohjelma näkisi parissa sekunnissa että 6.Ld2 onkin paha virhe (koska 6. - Rg4! ja musta voittaa) ja heti perään oikean siirron 6.Rf3 tasaisin asemin, mutta avauskirjastonsa pakottamana se pelaa miettimättä häviösiirron 6.Ld2 joka kerta. Ohjelma ei

läpäisisi Turingin testiä [Charmiak, 1984, s.10], sillä saman virheen toistaminen kerta toisensa jälkeen ei ole älykästä toimintaa. Kuitenkin jos tähän menetelmään lisättäisiin seuraavassa kappaleessa esitelty virheistä oppiminen, menetelmä olisi käyttökelpoinen.

Useimmat uudet ohjelmat osaavat pelaamiensa pelien perusteella oppia käyttämään avauskirjastoaan paremmin [Innala, 2001]. Perusversion sisältävät ohjelmat pyrkivät välttämään niitä avauksia, joissa niiden pelien lopputulokset ovat olleet heikkoja. Parannetussa versiossa ohjelma oppii välttämään niitä avausmuunnelmia, jotka varhaisessa keskipelissä johtavat ohjelman kannalta huonoihin aseman arvoihin.

4.1.2 Loppupelitaulukot

Loppupelissä nappuloiden käydessä vähiin on mahdollista hakea paras siirto suoraan taulukosta. Tällä hetkellä suurin loppupelitaulukko on kuuden nappulan loppupelitaulukko, joka sisältää kaikki mahdolliset enintään kuuden nappulan loppupeliasemat ja parhaimman siirron kussakin asemassa. Ongelmana on lähinnä loppupelitaulukkojen valtava tiedostokoko. Loppupelitaulukkoja ei ole täten vielä käytetty shakkiohjelmissa kovin laajalti, mutta tulevaisuudessa tiedon tallennuskapasiteetin kasvaessa loppupelitaulukkojen käyttö luultavasti yleistyy. Yksi nykypäivän vaihtoehto on hyödyntää tietokantapalvelimella sijaitsevaa loppupelitaulukkoa. Shakkiohjelma ottaa yhteyden tähän palvelimeen ja tiedustelee loppupeliaseman parhaimman siirron. Eräs tällaista palvelua tarjoava on Nalimov Tablebase Webservice.

4.2 Hakualgoritmit

Pelin alussa ohjelma seuraa avauskirjastoaan, mutta ennemmin tai myöhemmin ohjelma joutuu ulos avauskirjastostaan. Tässä vaiheessa ohjelman täytyy alkaa ajatella omilla "aivoillaan" ja tähän tarvitaan hakualgoritmia. Ensin hakualgoritmi rakentaa pelipuun kutsumalla siirtogeneraattorialiohjelmaa, joka käyttää jotakin luvussa 2 mainittua menetelmää. Pelipuun syvyys voi olla esimerkiksi 4 puolisiirtoa. Seuraavaksi hakualgoritmi kutsuu arvotusfunktioaliohjelmaa, joka arvioi pääteasemat jollakin luvussa 3 esitellyllä arvotusfunktiolla. Lopuksi hakualgoritmi suorittaa varsinaisen tehtävänsä, joka on löytää pelipuusta molemmin puolin paras mahdollinen jatko. Hakualgoritmien perustana on minimax-algoritmi, joka perustuu periaatteelle, jonka mukaan kumpikin valitsee kannaltaan parhaan vaihtoehdon, kun puuta käydään läpi alhaalta ylöspäin

[Hyvönen ym., 1993, s.180].

Hava innollistetaan asiaa käytännön esimerkillä. Oletetaan, että oltaisiin pelattu seuraavat siirrot.

1.e4 c5

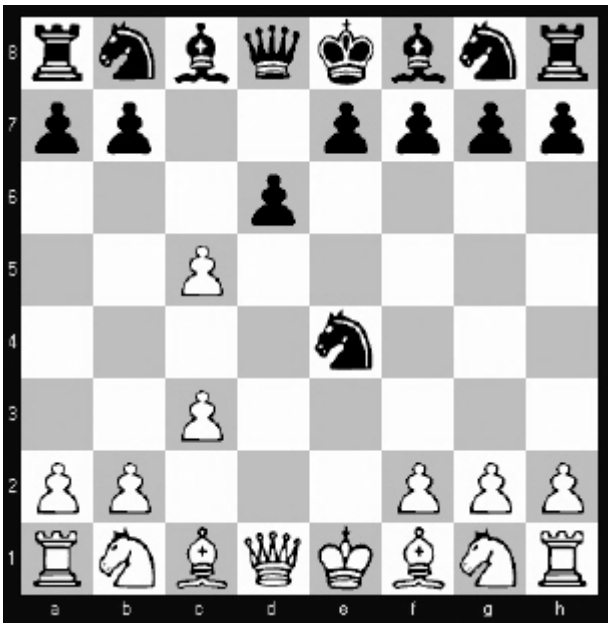
2.c3 Suosikkimuunnelmani sisilialaista puolustusta vastaan, c3 sisialialainen, jota kutsutaan myös Alapinin muunnelmaksi. Siirron perusidea on yksinkertaisesti muodostaa sotilaskeskusta jatkaen siirrolla d4.

2. - d6

3.d4 Rf6

4.dxc5 Rxe4? Inhimillinen virhe, johon pikapelissä ovat sortuneet minua vastaan mm. shakkimestari Tommi Luukkonen ja viimeisimpänä asema on ollut laudalla pikapelissä Aki Seppälä(2019)- Arto Holopainen(2053), 2004. Oikein olisi ollut teorian mukaisesti 4. - Rc6 (Nunn ym., 1999, s.159).

Katso kuvio 13.



Kuvio 13

Oletetaan nyt, että kone olisi pelannut valkeilla ja ihminen mustilla. Kone olisi tähän mennessä saanut siirrot suoraan avauskirjastostaan, mutta tässä kohtaa se joutuisi ulos avauskirjastostaan, sillä siellä on mainittuna vain aseman ainut teoriasiirto 4. - Rc6. Avauskirjastossa on toki joskus mainittu

huonojakin siirtoja, muttei tätä avausansaa, koska koneen on helppo löytää valkean voitto laskemalla vain kaksi siirtoa eteenpäin. Katsotaan kuinka tämä tapahtuu käytännössä.

Ensin siis muodostetaan pelipuu. Asetetaan pelipuun syvyydeksi kaksi siirtoa eli neljä puolisiirtoa. Valkealla on asemassa 34 mahdollista siirtoa, joten neljän puolisiirron päästä pelipuun lehtisolmuja on noin 34^4 eli yli miljoona. Jotta lehtisolmut mahtuisivat kuvaan, rajoitetaan näkymää niin, että näytetään vain kaksi ensimmäistä haaraa. Näin lehtisolmuja tulee $2^4 = 16$. Lisäksi pyritään saamaan kussakin asemassa kahden siirtovaihtoehdon joukkoon mahdollisimman monta shakkaavaa ja nappulan lyövää siirtoa, koska tällaiset voimakkaat siirrot ovat yleisesti potentiaalisia ehdokkaita parhaiksi siirroiksi

Seuraavaksi arvioidaan pelipuun kunkin lehtisolmun asema. Käytetään tähän yksinkertaista arvotusfunktiota, joka ottaa huomioon vain materiaalin ja liikkuvuuden. Määritetään liikkuvuuden painokertoimeksi maltillinen 0,03, joilloin arvotusfunktioksi saadaan:

aseman arvo = materiaali + 0,03 x liikkuvuus.

Lopuksi etsitään tästä pelipuusta molemminpuolin paras jatko minimax-algoritmilla.

Musta pyrkii minimoidaan aseman arvon ja valkea maksimoimaan, joten musta valitsee aina pienimmän mahdollisen ja valkea suurimman mahdollisen aseman arvon. Minimax-algoritmissa pelipuuta tutkitaan alhaalta ylöspäin, joten aloitetaan alhaalta. Käydään pelipuuta läpi vasemmalta oikealle. Ensimmäisenä musta valitsee jatkojen 6. - bxc6 ja 6. - Qd7 välillä. Musta valitsee aina pienimmän, joten musta valitsee jatkon, joka johtaa asema-arvoon -5. Musta valitsee edelleen oikealla kaikki vaihtoehdot valiten aina pienimmän mahdollisen arvon.

Musta minimoi syvyydessä 4:
min(-5,09, 3,21) min(2,39, 3,51) min (2,51, 3,96) min (-0,15, 0,03) min(0,21, 0,48) min(0,09, 3,27) min(4,88, 4,67) min(-2,48, 0,69)

Mustan minimoinnin tulokset syvyydessä 4:

- 5,09	2,39	2,51	-0,15	0,21	0,09	4,67	-2,48

Seuraavaksi siirrytään yksi taso ylöspäin. Nyt on valkean vuoro. Valkea maksimoi. Ensin valkea valitsee siirtojen 6.Dxc6+ ja 6.Dxe4 välillä. Valkea valitsee siirron 6.Dxe4, koska sillä siirrolla

valkea saa vähintään kahden sotilaan edun.

Valkea maksimoi syvyydessä 3:

max (-5,09, 2,39) max(2,51, -0,15) max(0,21, 0,09) max(4,67, -2,48)

Tulokset:

2,39	2,51	0,21	4,67

Mustan vuoro minimoida.

Musta minimoi syvyydessä 2:

Minointi:	min (2,39, 2,51)	min (0,21, 4,67)
Tulokset	2,39	0,21

Lopuksi valkea maksimoi syvyydessä 1:

Maksimointi:	max(2,39, 0,21)
Tulos:	2,39

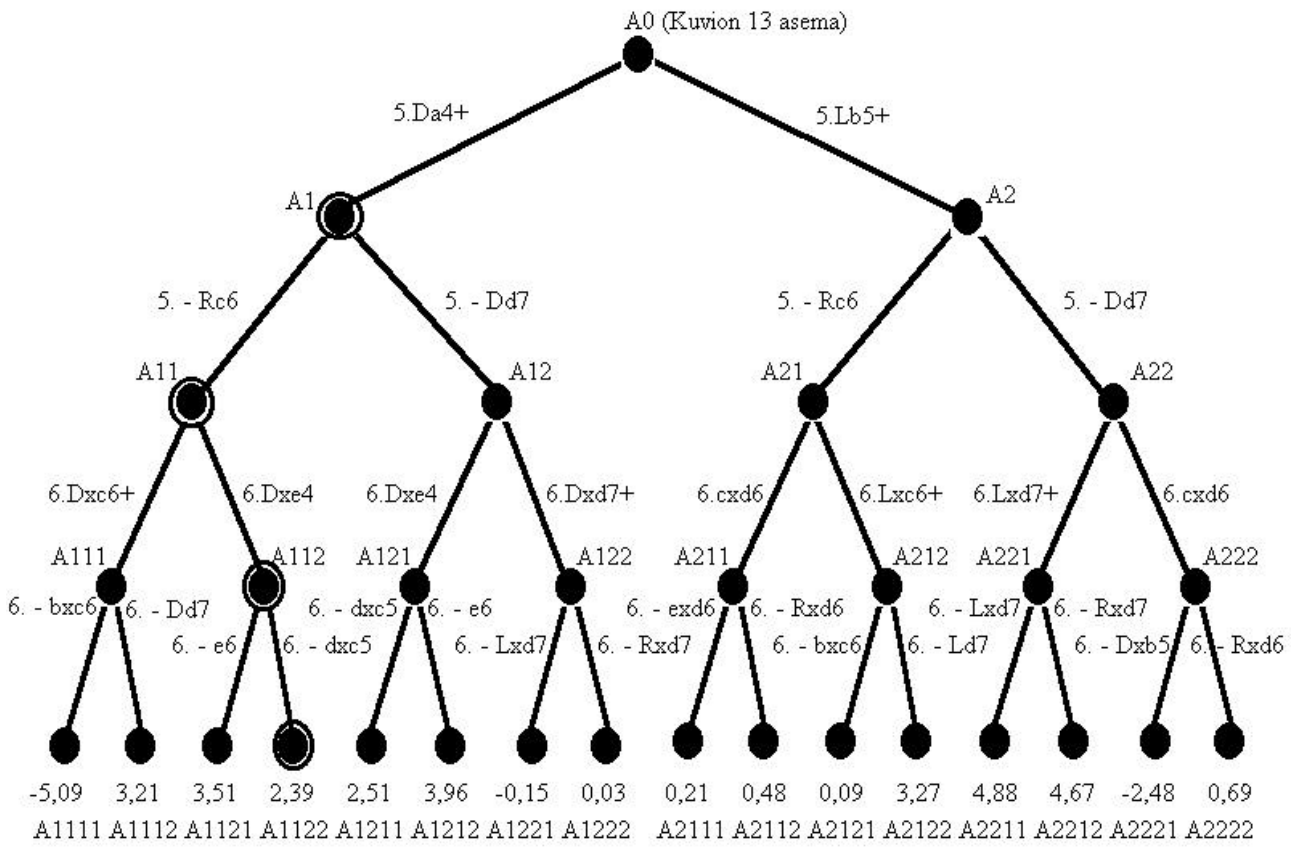
Nyt ohjelma tietää:

i) Paras siirto juuriasemassa on 1.Da4+

ii) Paras arvo minkä ohjelma voi saavuttaa molempien pelatessa parhaimpia siirtoja on 2,39.

iii) Molemmin puolin paras jatko on 1.Da4+ Rc6 2.Dxe4 dxc5

Katso kuva 4.



Kuva 4. Pelipuu minimax-haun jälkeen. Molemminpuolin paras jatko on merkitty ympyröimällä siihen kuuluvat asemat.

Tällä arvotusfunktiolla lopputulos ei muutu, vaikka mukana olisi kaikki loput yli miljoona jatkoa. Lopputulos on lisäksi myös shakillisesti varsin tarkka näinkin yksinkertaisella arvotusfunktiolla. Täsmälleen saman molemmin puolin parhaan jatkon antaa myös mm. Colossus Chess X. Arvotusfunktiolla, joka ottaisi huomioon vain materiaalin, löydettäisiin myös valkean voittosiirrot 1.Da4+ ja 2.Dxe4, mutta se ei erottaisi toisistaan esim. jatkoja 1.Da4+ Rc6 2.Dxe4 dxc5 ja 1.Da4+ Dd7 2.Dxe4 dxc5, koska molempien aseman arvo olisi sama 2. Arvotusfunktio, joka ottaa huomioon myös liikkuvuuden, tietää aivan oikein, että 2. - Rxc6 on asemallisesti hieman parempi mustalle kuin 2. - Dd7, koska jälkimmäisessä mustan daami sijaitsisi hieman kömpelösti tukkien oman lähetin diagonaalin.

Minimax-algoritmi ei tällaisenaan ole kovin käyttökelpoinen, koska jo kahden siirron syvyydelle menevässä haussa joudutaan käymään läpi yli miljoona asemaa ja kolmen siirron syvyydelle menevässä yli miljardi asemaa. Näin olleen tarvitaan menetelmiä tutkittavien asemien karsimiseen.

4.2.1 Minimax alfa-beta

Minimax alfa-beta-karsinnalla on ylivoimaisesti yleisin shakkiohjelmissa käytetty hakualgoritmi. Asemassa, jossa molemmilla pelaajilla on käytettävissään esimerkiksi shakin keskiarvoiset 37 siirtoa, alfa-beta-karsinnalla voidaan saavuttaa jopa 99,5 prosentin säästö eli vain joka 200:s asema tarvitsee tutkia [Hyvönen, 1993, s.181]. Alfa-beta-karsinnan nimi seuraa siitä, että siinä käytetään apuna kahta muuttujaa alfa ja beta. Alfa kuvaa parasta arvoa, minkä ohjelma voi saavuttaa. Beta puolestaan kuvaa huonointa arvoa, mihin ohjelma voi joutua. Haun alussa ohjelmalla ei ole mitään käsitystä molemmin puolin parhaan jatkon antamasta arvosta, joten alfa asetetaan äärettömäksi ja beta miinus äärettömäksi. Algoritmi etsii siirtoja, jotka kasvattavat betan arvoa (eli siirtoja, jotka ovat parempia ohjelmalle kuin paras siihen mennessä löydetty) tai vähentävät alfan arvoa. Kun ohjelma löytää siirron, joka antaa paremman aseman kuin mikään siihen mennessä löydetty siirto, se kasvattaa betan arvoa. Kun ohjelma löytää siirron, joka rajoittaa sen vastustajaa, se vähentää alfan arvoa. Lopullinen tulos, joka on alfan ja betan välissä, tarkentuu laskennan edetessä kun alfan ja betan arvot muuttuvat koko ajan lähemmäksi toisiaan.

Minimax alfa-beta-algoritmia käytetään yleensä depth first - periaatteella, mikä tarkoittaa, että ensin tutkitaan pelipuun vasemmanpuoleisin haara juuresta lehtisolmuun asti.

Tutkitaan kuinka algoritmi käytännössä toimii ja käytetään lähtöasemana jälleen kuvion 13 asemaa.

Ohjelma käy puuta läpi vasemmalta oikealle ja yrittää maksimoida tulostaan. Ensin se tutkii aseman A1111, minkä arvo on -5,09. Koska -5,09 on parempi ohjelmalle kuin paras tähän asti löydetty (miinus ääretön) ja parempi vastustajalle kuin ääretön, -5,09 on tällä hetkellä paras arvo minkä kumpikin voi saavuttaa. Seuraavaksi ohjelma tutkii aseman A1112, jonka arvo on 3,21. Tämä on parempi ohjelmalle, mutta huonompi vastustajalle, ja koska tässä on vastustajan siirto, ohjelma tietää, että vastustaja valitsee siirron 6. - bxc6, joka antaa asema-arvon -5,09.

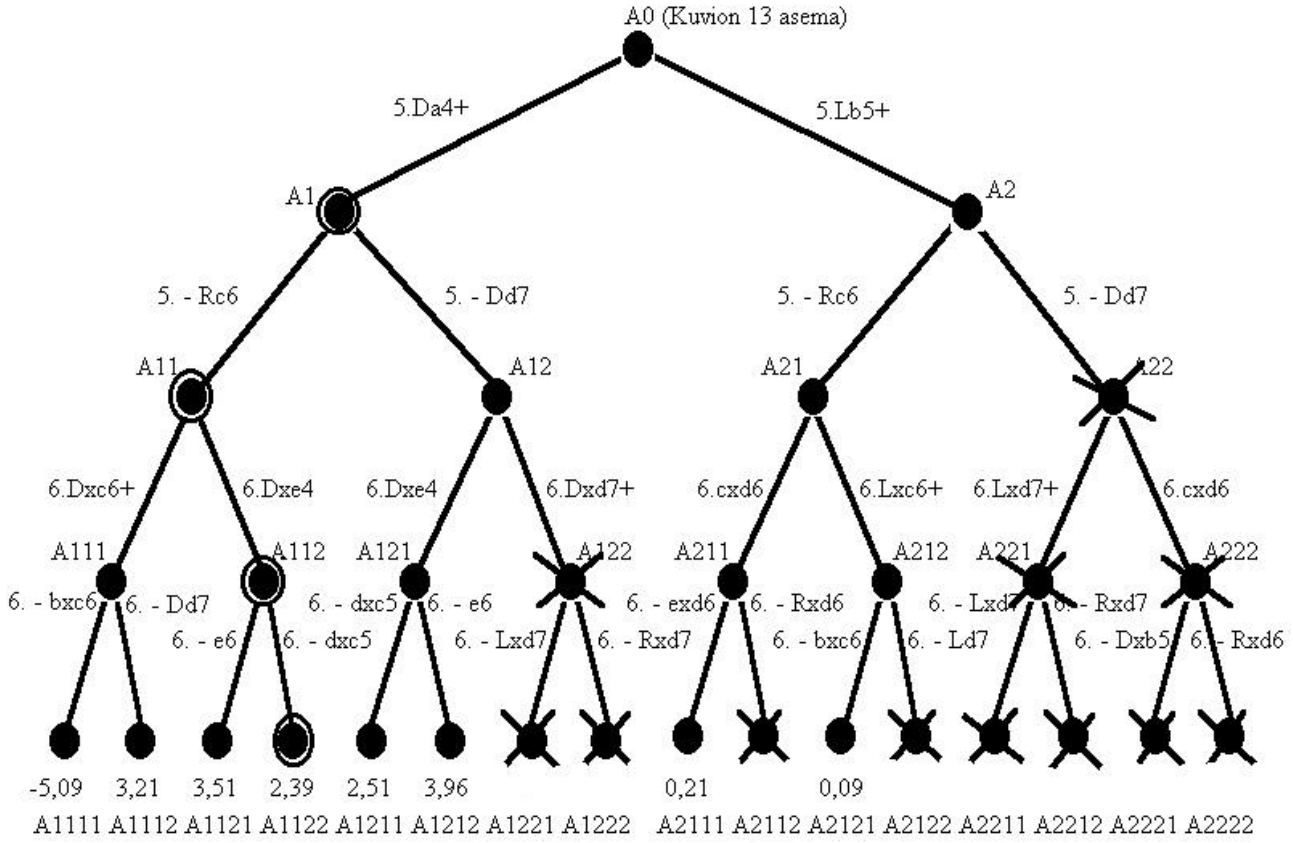
Seuraavaksi ohjelma tutkii aseman A1121, jonka arvo on 3,51. Tämä on ohjelmalle parempi kuin -5,09, joten se saavuttaisi mieluusti tämän aseman, mutta sen täytyy ensin tutkia muut ryhmän A112 asemat siltä varalta, että niistä jokin on parempi vastustajalle kuin -5,09. Aseman A1122 arvo on 2,39, joka on vastustajalle parempi kuin 3,51 ja parempi ohjelmalle kuin -5,09, joten tämä 2,39 on paras arvo, mikä voidaan saavuttaa asemasta A11 kummankin pelatessa oikein.

Ohjelma tutkii seuraavaksi siirtoa 5. - Dd7 ja asemia jotka, syntyvät aseman A12 jälkeen. Ohjelma

näkee, että jos se valitsee siirron 6.Dxe4 johtaen asemaan A121, niin silloin vastustajan on valittava asemien A1211, joka antaa arvon 2,51 ja aseman A1212, joka antaa arvon 3,96 välillä. Koska näistä kumpikin on vastustajalle huonompi kuin molemminpuolin paras tähän mennessä, joka on siis 2,39, ohjelma tietää, että vastustaja valitseee siirroista 6.- Rc6 ja 6.-Dd7 mieluummin siirron 6. - Rc6. Joten nyt ei tarvitse tutkia ollenkaan siirtoa 6. - Dxd7+, koska ohjelma tietää, että 6. - Rc6 on parempi vastustajalle kuin 6. - Dd7.

Ohjelma on nyt varma, että siirrolla 5.Da4+ se saa vähintään aseman, jonka arvo on 2,39. Ohjelma tutkii seuraavaksi voisiko se saada paremman aseman siirrolla 5.Lb5+. Ensin se tutkii aseman A2111, jonka arvo on 0,21. Koska 0,21 on huonompi ohjelmalle kuin 2,39 ja koska ohjelma tietää, että jos se valitsee siirron 6.cxd6 asemassa A21, vastustaja voi aina saavuttaa arvon 0,21, joka on ohjelmalle epäedullista, joten loppuja haaroja asemasta A211 ei tarvitse enää tutkia. Seuraavaksi se tutkii siirron 6.Lxc6+. Asema A2121 antaa arvon 0,09 ja koska 0,09 on huonompi ohjelmalle kuin 2,39, se ei halua antaa vastustajalle mahdollisuutta saavuttaa arvoa 0,09, joten ohjelma hylkää myös siirron 6.Lxc6+ eikä tutki sitä loppuun. Koska sekä 6.Lxc6+ että 6.cxd6 ovat ohjelmalle epäedullisia eikä muita vaihtoehtoja ole, ohjelma tietää, että vastustaja voi aina asemassa A2 valitsemalla siirron 5. - Rc6 saada itselleen paremman aseman kuin 2,39, joten alipuuta 5. - Dd7 ei tarvitse enää tutkia.
Nyt ohjelma tietää saman kuin normaalissa minimax-haussa, eli:
i) Paras siirto juuriasemassa on 1.Da4+
ii) Paras arvo minkä ohjelma voi saavuttaa molempien pelatassa parhaimpia siirtoja on 2,39.
iii) Molemmin puolin paras jatko on 1.Qa4+ Rc6 2.Dxe4 dxc5

Erona tavalliseen minimax-hakuun on se, että nyt algoritmin ei ole tarvinnut käydä läpi kaikkia asemia eikä laskea kaikkien lehtisolmuasemien arvoa arvotusfunktiolla. Tässä esimerkkitapauksessa arvotusfunktion käytössä saavutetaan 50 % säästö. Nyt tarvitsi arvioida vain 8 lehtisolmuasemaa, kun tavallinen minimax joutui arvioimaan kaikki 16 lehtisolmuasemaa. Katso kuva 5.



Kuva 5. Pelipuu mimax alfa beta-haun jälkeen. Rastilla merkittyjä asemia ei ole tarvinnut tutkia.

Minimax alfa-beta-karsinnalla antaa aina saman tuloksen kuin täyshaku normaalilla minimax-algoritmilla ja koska alfa-beta karsii huomattavasti tutkittavien asemien määrää, minimax alfa beta -algoritmi on toimiva ja tehokas. Algoritmia voidaan kuitenkin edelleen tehostaa muutamalla menetelmällä. Alfa-beta-ikkunat, Best-first haku ja siirtovaihtotaulukot alentavat läpikäytävien asemin määrää 35 % [Schaeffer, 1996].

Yksi tehostuskeinoista on siis alfa-beta-ikkunan käyttö. Tämä tarkoittaa, että sen sijaan että alfa ja beta aloittaisivat äärettömillä arvoilla, niille määritetään jokin kohtuullinen pieni arvo, kuten alfa = +2 ja beta = -2 ja tutkiaan vain asemia, joiden arvo on näiden välissä. Ideana on, että normaaliasemissa kumpikaan ei kummankin pelatessa parhaita siirtoja voi voittaa äkisti kovin paljon materiaalia. Algoritmi siis sivuuttaa monia epäoleellisia jatkoja ja säästytään turhalta laskennalta. Ongelmia tulee siitä, että menetelmä voi myös sivuuttaa jatkon, joka voittaisi pakottavasti suuren määrän materiaalia.

Best first-tehostuksessa pyritään aina ensin tutkimaan paras siirto. Alfa-beta-karsinta on tehokkaimillaan, jos paras siirto tutkitaan aina ensin. Tällöin tutkittavien asemien määrä on vain $2 \times d^b$, missä d on syvyys ja b keskimääräinen liikkuvuus [Levy, 1984, s.45]. Mahdolliset siirrot siis listataan paremmuusjärjestykseen ja algoritmille annetaan aina ensin listan ensimmäinen siirto. Siirtojen laittamiseen paremmuusjärjestykseen on monia tapoja. Kuten tämän luvun esimerkki osoitti, shakkaussiirrot ja nappulan lyövät siirrot kannattaa sijoittaa listan kärkeen. Vieläkin parempi keino on käyttää haussa iteratiivista syvennystä, jolloin voidaa hyödyntää tietoa edellisistä syvyyksistä. Jos siirto on ollut paras syvyydessä n-1, on hyvin todennäköistä, että on se on paras myös syvyydessä n.

Kolmas tehostuskeino on siirtovaihtotaulujen (transposition tables) käyttö. Kun ohjelma tutkii asemaa, se tallentaa kaikki läpi käymänsä asemat ja jos se joutuu myöhemmin haun aikana samaan asemaan eri siirtojen kautta, se säästää aikaa, koska aseman arvo on jo sen tiedossa eikä asemaa tarvitse enää tutkia uudelleen.

4.2.2 Heuristinen haku

Vaihtoehto minimax alfa-beta-algoritmille on heuristinen haku [Frey, 1978, s. 157 - 166]. Heuristisen haun idea pohjautuu vahvojen ihmispelaajien tapaan tutkia asema. Heuristinen haku ottaa huomioon vain hyvät siirrot.

Pelkistettynä heuristinen haku on minimax haku pelipuusta, joka sisältää vain "mielenkiintoisia" siirtoja. Siirtojen mielenkiintoisuus lasketaan arvotusfunktion perusteella.

Menetelmän hyvä puoli on, että hakupuu säilyy pienenä ja päästään syvemmälle. Huono puoli on, että pelipuu ei ole täydellinen, mikä voi johtaa siihen, että algoritmi ei löydä parasta mahdollista jatkoa. Esimerkiksi nappulan uhraukset, jotka myöhemmin antavat voittopelin, ovat heuristiselle haulle vaikeita löydettäviä.

4.3 Killer heuristiikka

Usein käytetty menetelmä, jolla voidaan lisätä kaikkien hakualgoritmien tehokkuutta, on nk. killer heuristiikka. Menetelmä perustuu nopean kumoussiirron olemassaoloon, joka kumoaa useimmat

siirrot.

Valaistaan asiaa muutamalla esimerkillä, jotka olen ottanut internetpikapeleistäni suurmestareita vastaan.

Tässä on ensimmäinen peli. Vastustajana on kolminkertainen USA:n mestari Larry Christiansen. Suluissa olevat ELO-luvut ovat kansainvälisiä pitkän pelin vahvuuslukuja. Katso kuvio 14.

Kuvio 14.

Aki Seppälä(ELO 1964) – Larry Christiansen (ELO 2531)

VALKEAN SIIRTO

Jos ohjelma alkaisi tutkia tätä asemaa, se saattaisi ensin mieltyä esimerkiksi siirtoon 1.Dd2, joka maksimoi valkean kuningattaren liikkuvuuden. Pian ohjelma kuitenkin huomaisi, että 1. - f5! antaa mustalle ratkaisevan edun. Se laittaisi siirron muistiin ja kokeilisi sitä muita valkean siirtoja vastaan. Killer heuristiikan nimi tulee siitä, että siirto 1. - f5! on tapposiirto (killer move). Haarukat (oma nappula uhkaa samaanaikaan kahta vastustajan nappulaa) ovat yleisiä tapposiirtoja. Tässä esimerkissä esiintyy sotilashaarukka, mutta ratsuhaarukka on vieläkin yleisempi. Tässä haarukan estäminen olisi kuitenkin helppoa. Mutta mustalla on joka tapauksessa sotilaan etu ja musta voittikin helposti.

Katsotaan vielä hieman monimutkaisempi esimerkki. Katso kuvio 15.

Kuvio 15.

Gadir Guseinov (ELO 2548) - Aki Seppälä (ELO 1964)

MUSTAN SIIRTO

Nuori azerbaidzanilainen suurmestari on uhrannut upseerin saadakseen tehtyä matin.

Jos ohjelma alkaisi tutkia tätä asemaa, se saattaisi ensin kokeilla siirtoa 1. - Lxf1. Mutta pian se huomaisi, että 2.Dg5+! Kh8 3.Dg7X tekee matin. Siis tässä asemassa valkean tapposiirto, joka mustan täytyy välttää on Dg5+. Pelissä musta näki tämän myös ja siirsi 1. - Kh8. Oletetaan taas, että ohjelma tutkisi jälleen asemaa. Se saattaisi yrittää siirtoa 2.c3, mutta nyt tappaja on 2. - Lxf1!. Valkea näki tämän myös ja teki ovelan (tosin hieman parempi, muttei riittävä, oli suoraan 2.Te7) vastauhkaussiirron 3.Dh6. Nyt on mustan vuoro ja on helppo nähdä, että valkean tapposiirto on nyt Dg7X. Musta siirsi siis 3. - Tg8, joka estää matin. Nyt on valkean vuoro ja mustan tapposiirto on jälleen Lxf1. Valkea torjui tämän vastauhkauksella 4.Te7!. Nyt on mustan vuoro. Valkean tapposiirto on tässä tietenkin 5.Txd7 ja peli jatkui 4. - Df5, joka riitti myös helposti mustalle voittoon. Itseasiassa hieman tarkempi olisi ollut 4. - Dg4!, joka olisi asettanut valkean kiusaksi jälleen tapposiirron. Nyt olisi uhannut 5. - Dxg2X ja peli olisi voinut jatkua 6.g3 Lxf1 ja musta olisi saanut vielä kvaliteetinkin. Mutta kuten sanottua upseerin etukin takasi mustalle helpon voiton (0-1).

Kuten edellä olevat esimerkit osoittavat käytännön peleissä esiintyy usein tapposiirtotilanteita ja niiden tunnistaminen vähentää laskemisen tarvetta.

Killer heuristiikka voidaan implementoida ohjelmaan eri tavoilla. Yleisin tapa on tallentaa jokaisesta syvyydestä yksi tapposiirto ja kokeilla sitä ensiksi seuraavassa syvyydessä. Jos tapposiirto ei enää toimi tässä syvyydessä, sen tilalle otetaan uusi kumoussiirto. Toinen lähestymistapa on tallentaa jokaisessa syvyydessä useampi tapposiirto. Siirtolista järjestetään sen perusteella, kuinka usein siirto on ollut tapposiirtona. Ohjelma kokeilee sitten ensin siirtoja listan kärjestä. On olemassa vielä kolmaskin tapa. Ylläpidetään tapposiirtoa tai tapposiirtolistaa jokaista mahdollista vastustajan siirtoa vastaan.

4.4 Horisonttiongelma

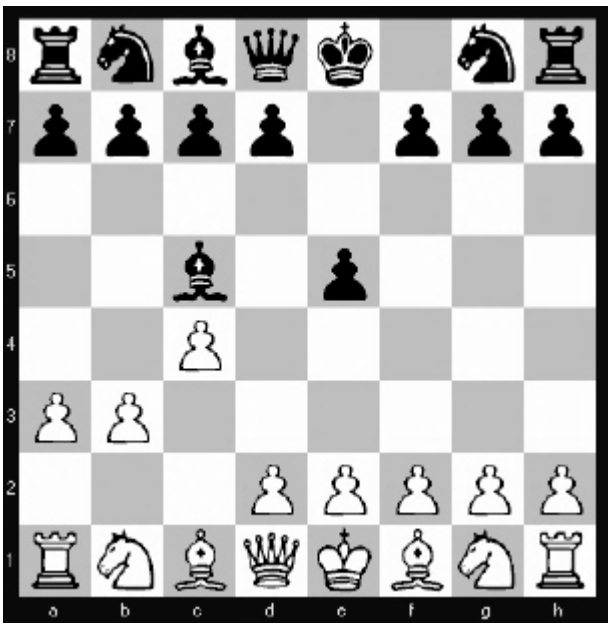
Yksi hakualgoritmien suurimpia ongelmia on horisonttiongelma. Se tarkoittaa, että ohjelma tekee virheen, koska ei näe tarpeeksi pitkälle. Otetaan muutama esimerkki.

Suurmestari Yrjö Rantanen on kehittänyt seuraavan näköhorisonttiavausansan, jota hänen oli tarkoitus käyttää Aamu-TV-pelissä konetta vastaan. Hän ei kuitenkaan päässyt käyttämään ansaansa TV-kameroiden edessä, koska kone sattui silloin pelamaan eri muunnelmaa ja Rantanen hävisi. Mutta katsotaan millainen ansa hänellä oli mielessä.

Muunnelma alkaa

1.c4 e5
2.b3 Lc5 Rantanen toivoi tätä lähetin siirtoa, mutta kone pelasikin toisen siirron.
3.a3!? Virittää ansan koneille.

Katso kuvio 16.



Kuvio 16

Shakkiohjelmat pelaavat tässä asemassa yleensä 3. - Lxf2+?, koska ne luulevat voittavansa materiaalia. Ne eivät näe, että musta menettää kuningattaren, koska se on useimpien ohjelmien laskuhorisontin ulkopuolella. Tähän ansaan lankeaa myös mm. Rebel 10.

Katsotaan kuinka ansa toimii.

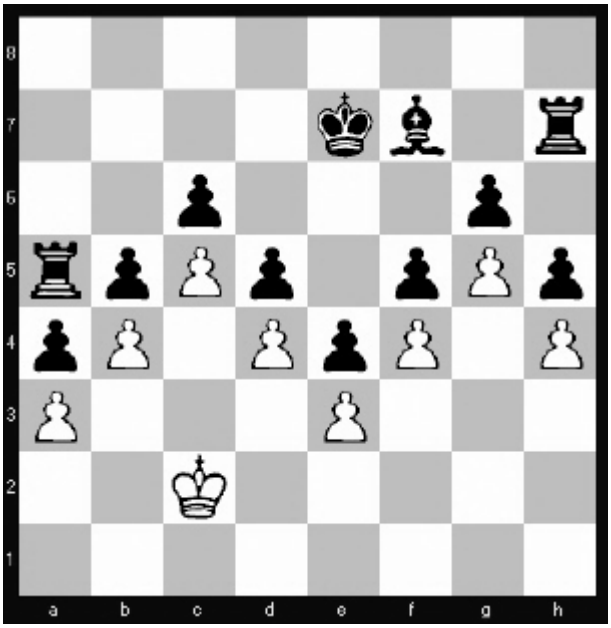
3. - Lxf2+? Ratkaiseva virhe.
4.Kxf2 Dh4+
5.g3 Dd4+
6.e3 Dxa1
7.Dc2 Rf6
8. Lb2 Re4+
9. Kg2 Da2
10.Re2

Katso kuvio 17.

Kuvio 17.

Valkea pelaa seuraavaksi 11.Rc1 ja musta menettää daaminsa. Näin pelin alussa tapahtuva ansa on helppo välttää avauskirjastoa muokkaamalla, mutta samantyyppinen ansa myöhemmin pelissä on shakkiohjelmille iso haaste.

Hartston esittää artikkelissaan "Artificial stupidity" [Beal, 1986, s. 55] asemia, jotka ovat koneelle ylivoimaisen vaikeita, mutta ihmiselle helppoja. Yksi asemista on seuraavanlainen. Katso kuvio 18.



Kuvio 18

VALKEAN SIIRTO

Nykyisillä ohjelmilla ei ole edellytyksiä ratkaista tämänkaltaisia ongelmia. Ihminen näkee ensisilmäyksellä, että siirtämällä kuningasta edestakaisin valkea saa tasapelin, koska mustalla ei ole mitään tapaa edistyä. Kone sen sijaan ei tätä näe, vaan pelaa 1.bxa5, kuten myös Rebel - ohjelma tekee, mutta tällöin musta voittaa, koska mustan b4-sotilas vapautuu nyt etenemään ja valkean puolustusmuuri hajoaa. Jotta ohjelma näkisi tasapelin, sen pitäisi laskea 50 siirron päähän, jolloin tasapeli tulisi automaattisesti 50 siirron säännön takia.

Pohditaan lopuksi eri tapoja, joilla horisonttiongelmaa voisi helpottaa.

4.4.1 Valikoiva jatkolaskenta

Yleisesti tehdään niin, että muunnelmia, jotka jatkuvat lehtisolmuasemista shakeilla tai nappulan lyövillä siirroilla, lasketaan pidemmälle kuin muita asemia. Tätä kutsutaan valikoivaksi jatkolaskennaksi. Tavoitteena on, että saavutetaan hiljainen asema, jolloin aseman arvo on luotettavampi. Aina tätä tavoitetta ei kuitenkaan voi saavuttaa, koska esimerkiksi shakkeja voi olla niin paljon, että ohjelma kuluttaisi aikansa loppuun niitä tutkiessaan, joten yleensä jatkolaskennan rajaksi asetetaan jokin tietty syvyys, jonka yli jatkolaskentaakaan ei saa tehdä.

4.4.2 Pysyvät muistitaulukot

Pysyviä muistitaulukkoja ei käytä vielä juuri mikään ohjelma, mutta ainakin Nimzo 8 käyttää [Innala , 2001]. Ideana on, että ohjelma tallentaa pysyvästi kaikki pelaamansa asemat ja myös käyttäjän kanssa analyysin yhteydessä esille tulleet asemat ja osaa näin ajoissa välttää vaikeuksia. Myös Shredder4 sisältää mielenkiintoisen oppimistoiminnon. Se tallentaa sellaiset asemat, joissa ilmenee merkittäviä lyhyen tähtäimen aseman arvon muutoksia eli tilanteita, jotka ohjelma on alun perin arvioinut väärin. Pysyvien muistitaulukkojen huono puoli on, että niistä on hyötyä vain silloin, kun ohjelma pääsee uudestaan täsmälleen samanlaiseen asemaan kuin missä se on jo ollut. Ensimmäisellä kerralla apua on vain valikoivasta jatkolaskennasta.

4.4.3 Omia ideoita

Minua on mietittänyt eräs asia. Kun ohjelma miettii jatkoa, siirrot eivät juurikaan vaihdu syvemmälle mentäessä ja pian laskenta jymähtää paikalleen hakupuun kasvaessa valtavaksi. Kuitenkin jos käyttäjä keskeyttää laskennan ja antaa koneen tehdä muutaman siirron pitkän muunnelmansa alusta niin ohjelma saattaa nähdä parissa sekunnissa jotain ratkaisevaa, kuten että jatko häviääkin pakottavasti. Mieleen tulee, että eikö tämän voisi tehdä myös ohjelmallisesti. Kone tuhlaa laskuvoimaansa valtavaksi haaroittuneen pelipuun tutkimiseen, kun voisi katkaista puun sopivasta kohdasta, johon kasvattaa uusi pelipuu ja tarkistaa ollaanko selvillä vesillä.

Yksi mahdollisuus olisi tarkistaa jokainen pelattavaksi aiottava jatko kunnollisella monen siirron jatkolaskennalla lähtökohtana joko alkuperäisen laskennan lehtisolmu tai siitä vähän peruuttaen. Jos nyt havaitaan, että koneen asema huonontuu merkittävästi, tulisi tarkistaa samanlaisella jatkolaskennalla olisiko esimerkiksi alkuperäisen laskennan toiseksi paras siirto sittenkin parempi. Itse asiassa Deep Bluessa on jo sovellettu hieman tämän tapaista menetelmää. Se tarkistaa pelattavaksi aikomansa pelipuun haaran laskemalla sitä syvemmälle kuin muita siirtovaihtoehtoja. Tosin laskennan lähtökohtana on vain puolisiirron syvempänä oleva asema, joten tämäkin laskenta puuroutuu hyvin nopeasti hakupuun levitessä. Paljon syvemmälle päästäisiin, jos otettaisiin jatkolaskennan lähtökohdaksi joku paljon myöhäisempi asema.

Radikaalimpi mahdollisuus olisi muuttaa koko hakustrategia. Voitaisiin esimerkiksi ensin hakea syvyydelle n, sitten peruuttaa syvyyteen n/2, josta uusi haku syvyydelle n, ja taas peruutetaan syvyydelle n/2, johon kasvatetaan taas uusi pelipuu ja niin edelleen. Peruutuksen idea on, että oletetun jatkon puolivälissä ollaan varmemmalla pohjalla kuin ihan muunnelman lopussa. Tämän menetelmän huono puoli on, että pelipuu ei ole täydellinen ja joitakin pitkiä voittojatkoja voi jäädä näkemättä, mutta toisaalta kone pelaisi kokoajan kohtalaisen hyviä siirtoja ja näkisi valtavan pitkälle verrattuna normaaliin hakuun. Tilanteissa, joissa kone joutuu normaalisiirroilla pitkän siirtosarjan päässä häviöasemaan (esimerkiksi jokin nappula on jäänyt ansaan) eli tyypillisissä horisonttiongelmatilanteissa tästä menetelmästä olisi hyötyä.

Ihminen analysoi usein pelaamansa pelin ja oppii virheistään. Samoin voisi tehdä kone. Kone voisi itse yrittää syväanalysoida pelinsä pelin jälkeen ja etsiä täsmällisesti mikä oli huono siirto ja mikä olisi ollut parempi ja muistaa nämä myös jatkossa. Tällä hetkellä on olemassa vain menetelmä,

jossa ohjelma oppii käyttäjän ohjaaman analyysin aikana ja toinen menetelmä, jossa kone oppii välttämään tiettyjä muunnelmia, mutta se ei tiedä tarkalleen, missä kohtaa huono siirto on. Se voi siis esimerkiksi lähteä johonkin vaatimattomaan muunnelmaan, kun vaihtoehtoisessa paremmassa terävässä muunnelmassa se on tehnyt jonkin virheen, johon olisi kuitenkin löydettävissä parannus.

Kuvion 18 kaltaiset asemat ovat koneelle erittäin vaikeita ymmärrettäviä. Luultavasti yksikään ohjelma ei vieläkään osaa käsitellä tämän tapaisia asemia oikein. Ongelma on vaikea, mutta ei mahdoton. Yksi mahdollisuus olisi käyttää edellä mainittua radikaalia lineaarista hakustrategiaa. Toinen mahdollisuus olisi yrittää opettaa koneelle, että tietynlaiset suljetut asemat ovat tiettyjen ehtojen vallitessa kuolleita tasapelejä ja asettaa tällaisten asemien arvotusfunktion arvo nollaan, jolloin se ei lähde häviöasemassa rikkomaan asemaansa. Siis lisätään shakkitietämystä, eikä yritetäkään laskea muunnelmia loppuun.

5 SOVELLUSESIMERKKI

Edellisissä luvuissa on kerrottu, kuinka shakkiohjelmalle teoriassa ohjelmoidaan tekoäly. Tässä luvussa siirrytään käytäntöön, ja selostan, kuinka ohjelmoin perusshakkiohjelman.

5.1 Lähtökohdat

Tarkoituksena on siis toteuttaa perustekniikoilla shakkia pelaava ohjelma. Lähtökohtana ovat Shannonin [Shannon, 1950] alkuperäiset ideat ja muut edellisissä luvuissa esitetyt perustekniikat. Ohjelmointiin oli aikaa vain kaksi viikkoa, joten graafisen käyttöliittymän tekemiseen ei ollut aikaa, mutta se ei alunperinkään ollut pääasia, vaan tekoälyn ohjelmointi, kuten tutkielman otsikkokin sanoo.

5.2 Toteutus

Valitsin ohjelmointikieleksi C++:n ja ohjelmointityökaluksi Borland C++ 3.1:n. Seuraavassa selostan, kuinka ohjelmointi eteni, ja esitän myös keskeisimpiä koodirivejä. Koko koodi on liitteessä 1.

5.2.1 Laudan representaatio ja sallittujen siirtojen tuottaminen

Sovelsin lukujen 2.1 ja 2.2 menetelmiä siten, että laudan representaatio on luvusta 2.2, mutta siirtojen tuottaminen on luvusta 2.1.

Lautana toimii 12x12-taulukko, ja nappuloiden arvot ovat kuten luvussa 2.2.

```
int lauta[12][12] = {{99,99,99,99,99,99,99,99,99,99,99,99},
                     {99,99,99,99,99,99,99,99,99,99,99,99},
                     {99,99,-4,-2,-3,-5,-6,-3,-2,-4,99,99},
                     {99,99,-1,-1,-1,-1,-1,-1,-1,-1,99,99},
                     {99,99,0,0,0,0,0,0,0,0,99,99},
                     {99,99,0,0,0,0,0,0,0,0,99,99},
```

```
{99,99,0,0,0,0,0,0,0,0,99,99},
{99,99,0,0,0,0,0,0,0,0,99,99},
{99,99,1,1,1,1,1,1,1,1,99,99},
{99,99,4,2,3,5,6,3,2,4,99,99},
{99,99,99,99,99,99,99,99,99,99,99,99},
{99,99,99,99,99,99,99,99,99,99,99,99}};
```

Siirtojen tuottaminen tapahtuu kuitenkin rivi- ja sarakepohjaisesti. Etuina ovat, että nyt pärjätään yhdellä laudalla ja siirtojen tuottaminen on havainnollisempaa.

```
if(lauta[rivi-1][sarake-2] != 99 && ((lauta[rivi-1][sarake-2]<1&&siirtovuoro==1)
    || (lauta[rivi-1][sarake-2] > -1 && siirtovuoro==-1)))
      LisaaSiirto(alku,rivi,sarake,rivi-1,sarake-2,0);
```

Luvun 2.2 menetelmällä kyseinen ratsun siirto tuotettaisiin operaatiolla +8, josta ei ihan helposti ilmene shakin säännöt. Nyt sen sijaan shakin sääntöjäkin osaamaton voisi pysyä mukana eli tässä tapauksessa ratsu haluaisi siirtyä yhden ruudun eteenpäin (rivi-1) ja kaksi ruutua vasemmalle (sarake-2). Ensin on kuitenkin tarkistettava, onko kohderuutu laudalla (ruudun arvo ei saa olla 99) ja ettei kohderuudussa ole omaa nappulaa, sillä omaa nappulaa ei tietenkään voi syödä. Jos siirtovuoro on 1, on valkean vuoro, ja tällöin tarkistetaan, ettei ruudun arvo ole positiivinen, joka tarkoittaisi, että ruudussa on jo oma nappula. Jos siirtovuoro on -1, on mustan vuoro, ja tällöin tarkistetaan ettei ruudun arvo ole negatiivinen. Itse asiassa siirtojen tuottamisessa opetetaan ohjelmalle shakin säännöt, ja periaatteessa tätä koodia tutkimalla shakin säännöistä mitään tietämätön ihminen voisi oppia ne.

Kun siirto on tuotettu, se lisätään dynaamiseen linkitettyyn siirtolistaan. Jokaisesta siirrosta talletetaan alkurivi, alkusarake, kohderivi, kohdesarake ja korotusnappula, jota tarvitaan, jos sotilas korottuu, jolloin voidaan valita, mikä nappula sotilaasta halutaan korottaa.

Tämä menetelmä sopii hyvin kenelle tahansa, joka tekee ensimmäisen shakkiohjelmansa, koska menetelmä on havainnollinen ja helppo ohjelmoida. Menetelmä ei kuitenkaan ole tehokas, sillä laskutoimituksia ja muistia tarvitaan enemmän kuin esim. luvun 2.2 menetelmässä. Nyt tarvitaan laskutoimitus sekä riviä että saraketta kohti ja muistia on varattava viidelle kokonaisluvulle, kun vastaavasti luvun 2.2 menetelmässä pärjätään yhdellä laskutoimituksella (esim. edellä mainittu +8) ja kolmella kokonaisluvulla (lähtöruutu, kohderuutu ja korotusnappula). Kun haluaa kehittää ohjelmastaan tehokkaamman, kannattaa vaihtaa siirtojen tuottaminen kehittyneempään menetelmään.

5.2.2 Perusmateriaalifunktio

Kun siirrot tuottava osa oli valmis, seuraavaksi tein yksinkertaisen arvotusfunktion, joka otti huomioon vain materiaalin. Siis käydään lauta läpi, ja vähennetään aseman arvoa, jos löydetään musta nappula, ja kasvatetaan aseman arvoa, jos löydetään valkea nappula. Jos löydetään esimerkiksi luku -2, tiedetään kyseessä olevan musta ratsu, ja aseman arvoa vähennetään kolmella, joka on ratsun arvo sotilaina.

```
double ArvioiAsema(int lauta[12][12)
{
  double asemanArvo = 0.00;
  int rivi, sarake;
  for (rivi = 2; rivi < 10; rivi ++)
    for (sarake = 2; sarake < 10; sarake++)
    {
      if (lauta[rivi][sarake]==-1)
        asemanArvo=asemanArvo-1;
      if (lauta[rivi][sarake]==-2)
        asemanArvo=asemanArvo-3;
       …

return asemanArvo;
}
```

5.2.3 Välivaiheen hakualgoritmi

Seuraavaksi tein hakualgoritmin. Se on luvun 4.2.1 mukainen puhdas minimax alfa beta. Seuraavassa on esitetty välivaiheen hakualgoritmin koodin keskeisimmät kohdat.

```
double MiniMaxAlfaBeta(int syvyys,double alfa,double beta, ...)
{
  if (syvyys == 0) {
    return ArvioiAsema(lauta);
}

  TuotaKaikkiMahdollisetSiirrot(lauta,siirtovuoro,...);

  kohdalla = laillistenSiirtojenAlku;
  while (kohdalla)
  {

    TeeSiirto(kohdalla...);
    asemanArvo = -MiniMaxAlfaBeta(syvyys-1, -beta, -alfa, ...)
    KumoaSiirto(kohdalla...);

    if (asemanArvo >= beta) {
      return beta; }

    if (asemanArvo > alfa) {
      alfa = asemanArvo;
```

```
      if (asetettuSyvyys == syvyys) {
        parasSiirto[0] = kohdalla->alkuRivi;
        parasSiirto[1] = kohdalla->alkuSarake;
        parasSiirto[2] = kohdalla->kohdeRivi;
        parasSiirto[3] = kohdalla->kohdeSarake;
        parasSiirto[4] = kohdalla->korotusNappula;
      }
    }
    kohdalla = kohdalla->seuraava;
  }
return alfa;
}
```

Ensin tuotetaan juuriaseman kaikki mahdolliset siirrot (TuotaKaikkiMahdollisetSiirrot). Mahdolliset siirrot ovat muuten korrekteja, mutta kuninkaan jäämistä syötäväksi ei oteta huomioon. Sitten käydään siirtolista läpi alusta loppuun. TeeSiirto-aliohjelma tekee listan siirron laudalle. Seuraavaksi kutsutaan itseään (MiniMaxAlfaBeta) niin kauan kunnes saavutetaan lehtisolmu eli syvyys 0. Miinusmerkeillä hoidetaan se, että algoritmi vuoronperään minimoi ja maksimoi. Lehtisolmut arvioidaan ArvioiAsema-aliohjelmalla. Sitten aletaan pikkuhiljaa peruuttaa kohti juurisolmua, ja lopussa talletetaan paras alkusiirto parasSiirto-taulukkoon ja palautetaan palautusarvona molemmin puolin parhaan jatkon lehtisolmun arvo.

Alfa-beta-karsinta on helppo toteuttaa, sillä siihen ei tarvita kuin kaksi riviä:

```
if (asemanArvo >= beta)
  return beta;
```

Tätä kutsutaan perinteisesti beta-katkaisuksi (beta cutoff) [Norvig, 1992, s.615]. Tässä vaiheessa ohjelma maksimoi, ja jos se löytää samalta syvyydeltä eri haarasta aseman, joka on parempi kuin arvo edellisessä parhaassa haarassa, niin ohjelma tietää, että vastustaja voi aina välttää tämän paremman arvon minimoimalla seuraavassa syvyydessä, eikä puuta tarvitse tutkia tästä eteenpäin.

Kun MinimaxAlfaBeta-aliohjelmaa kutsutaan seuraavan kerran, minimoidaan, ja tällöin edellinen beta-katkaisu kääntyy kätevästi suoraan alfa-katkaisuksi:

```
if (asemanArvo <= alfa)
  return alfa;
```

Jos vastustaja voi saavuttaa tässä haarassa huonomman arvon koneen kannalta kuin paras tähän asti kokonaan tutkittu haara-arvo koneen kannalta, niin tämä haara voidaan katkaista.

Tässä kohtaa pelasin ensimmäisen pelin ohjelmaani vastaan.

Aki Seppälä – Perusshakki

1.e4 Ra6??
2.Lc4 Tb8?
3.Dh5 Ta8??
4.Dxf7X (Kxf7
5.Lxf7)

Ohjelman kaikki siirrot olivat todella huonoja, vaikka ohjelma laski kolmen puolisiirron syvyyteen asti. Ratsun siirto oli lähes huonoin mahdollinen aloitussiirto, tornin siirto täysin turha ja koulumatin salliminen alokasmaista. Lisäksi peli loppui vasta, kun olin syönyt kuninkaan.

Ensimmäisen siirron ongelma oli helppo hoitaa lisäämällä arvotusfunktioon liikkuvuus, jolloin ohjelma siirsi mieluummin Rc6 kuin Ra6, sillä c6:ssa ratsulla on paljon enemmän siirtomahdollisuuksia kuin a6:ssa.

Ohjelma meni kuitenkin edelleen koulumattiin, enkä ollut alkuunkaan tyytyväinen, joten parannuksia tarvittiin.

5.2.4 Laillisten siirtojen tuottaminen

Nyt tuli ajankohtaiseksi laillisten siirtojen tuottaminen. Tähän käytin Shannonin [Shannon, 1950] alkuperäistä ideaa. Tehdään siirto listasta ja tutkitaan jäikö kuningas siihin. Tutkimisen hoitaa TarkistaLaillisuus-aliohjelma, joka tekee vastustajan kaikki mahdolliset siirrot, ja ilmoittaa, että sille annettu siirto oli laiton, jos joku näistä vastustajan siirroista kohdistuu kuninkaaseen.

Tämän jälkeen oli helppo lisätä matin ja patin tunnistus. Se on lisättävä sekä arvotusfunktioon että myös itse hakualgoritmiin, jolloin ohjelma tunnistaa matin ja patin myös hakupuun keskellä. Tunnistus tehdään käytännössä siten, että jos siirtovuorossa olevalla pelaajalla ei ole siirtoja eli laillisten siirtojen lista on tyhjä, pelaaja on joko matissa tai patissa. TarkistaOnkoMatti-aliohjelma tutkii kummasta tapauksesta on kyse tuottamalla kummankin osapuolen mahdolliset siirrot. Kun siirrot on tuotettu, tiedetään onko tilanne matti vai patti. Jos kuningas on uhattuna, kuningasta uhkaava osapuoli on tehnyt matin. Jos taas kumpikaan kuningas ei ole uhattuna, tilanne on patti.

5.2.5 Laillisuuden huomioiva hakualgoritmi

Laillisten siirtojen tuottamisen jälkeen hakualgoritmi jalostui seuraavanlaiseksi:

```
double MiniMaxAlfaBeta(int syvyys,double alfa,double beta, ... )
{
  if (syvyys == 0) {
    return ArvioiAsema(lauta);
  }

   TuotaKaikkiLaillisetSiirrot(lauta,siirtovuoro, ... )
   if (laillistenSiirtojenAlku == NULL)
     return TarkistaOnkoMatti(lauta);
  kohdalla = laillistenSiirtojenAlku;
  while (kohdalla)
  {
    TeeSiirto(kohdalla->alkuRivi, kohdalla->alkuSarake,kohdalla-
    asemanArvo = -MiniMaxAlfaBeta(syvyys-1, -beta, -alfa, parasSiirto,
    KumoaSiirto(kohdalla->alkuRivi, kohdalla->alkuSarake, kohdalla->kohdeRivi,
    if (asemanArvo >= beta)
      return beta;

    if (asemanArvo > alfa) {
      alfa = asemanArvo;
      if (asetettuSyvyys == syvyys) {
      parasSiirto[0] = kohdalla->alkuRivi;
      parasSiirto[1] = kohdalla->alkuSarake;
      parasSiirto[2] = kohdalla->kohdeRivi;
      parasSiirto[3] = kohdalla->kohdeSarake;
      parasSiirto[4] = kohdalla->korotusNappula;
      }
    }
    kohdalla = kohdalla->seuraava;
  }
  return alfa;
}
```

Verrattuna aiempaan luvun 5.2.3 versioon ohjelma harkitsee nyt vain laillisia siirtoja ja katkaisee hakupuun matin ja patin kohdilta, jolloin eteenpäin välittyy joko matin arvo tai patin arvo. Matin arvo on 1000, jos valkea on tehnyt matin, -1000, jos musta on tehnyt matin ja 0, jos tilanne on patti.

Haittapuolena on, että ohjelman suoritus hidastuu merkittävästi, mutta nyt ohjelma pelaa laillista shakkia ja tunnistaa matin ajoissa. Esimerkiksi ohjelma ei enää mennyt koulumattiin kolmen puolisiirron syvyydessä eikä kahdenkaan puolisiirron syvyydessä.

5.2.6 Arvotusfunktion osat

Lopuksi lisäsin arvotusfunktioon lisää osia materiaalin ja liikkuvuuden lisäksi. Tarkoituksena oli toteuttaa kukin osa mahdollisimman yksinkertaisesti, mutta myös käytännössä toimivasti.

Keskustan hallinta määritetään yksinkertaisesti siten, että jokaisesta keskustaruudun e4, d4, e5 ja d5 hallinnastaa saa pienen bonuksen.

Kehitysfunktiona oli aluksi luvun 3.3.1 yksinkertainen kehitysfunktio, mutta se ei toiminut kovinkaan hyvin käytännössä, koska ohjelma siirsi mielellään kuningatarta heti alussa maksimoidakseen liikkuvuutensa. Niinpä lisäsin kehitysfunktiooni luvun 3.3.2 kehittyneestä kehitysfunktiosta kuningattaren aikaista kehittämistä rajoittavan osan. Nyt kehitys toimi hyvin, ja ohjelma tajusi toisinaan jopa linnoittaa, vaikken ollutkaan lisännyt sitä osaa kehittyneestä kehitysfunktiosta.

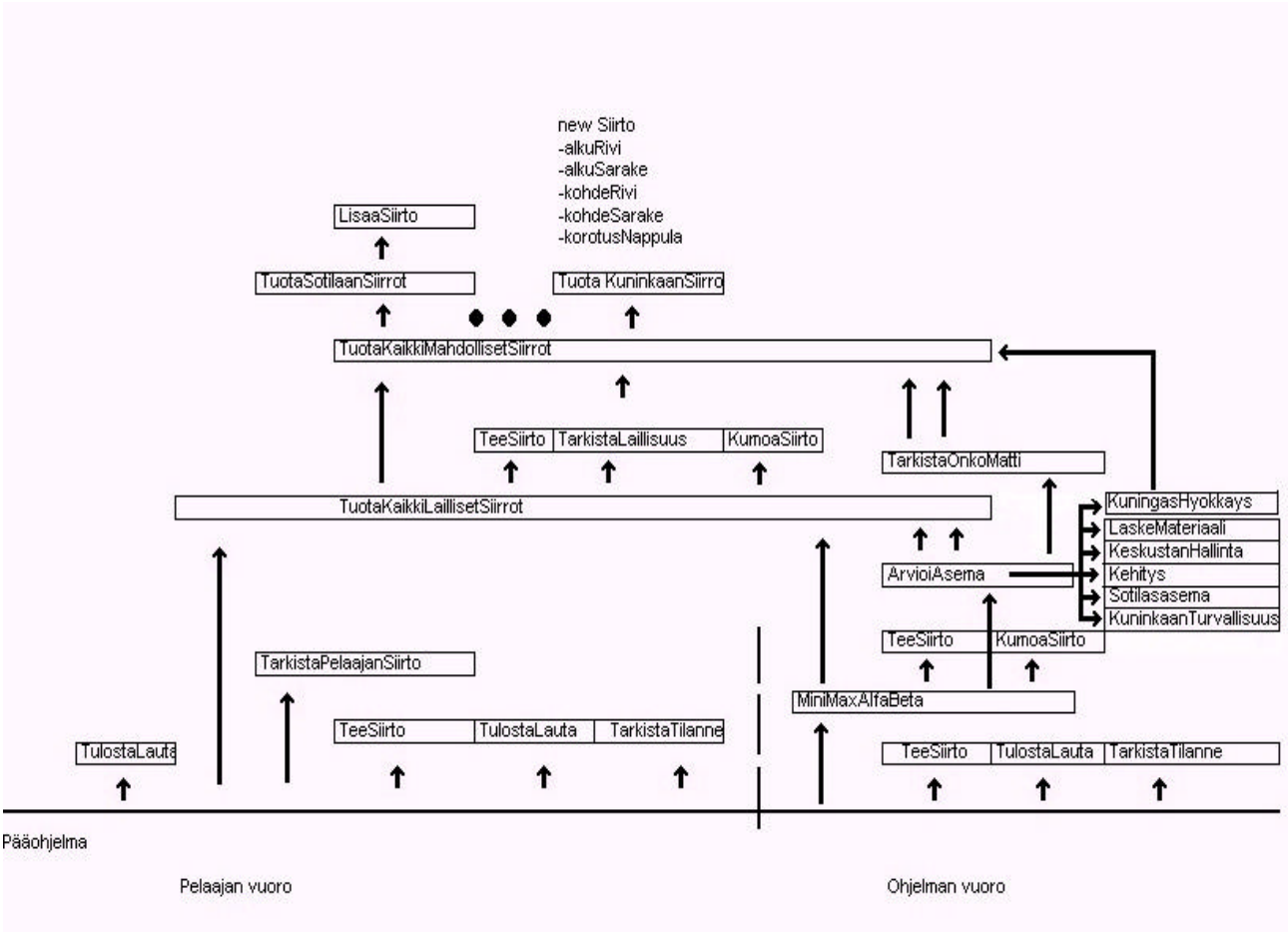
Sen sijaan olin lisännyt kuninkaan turvallisuuden, jonka päätarkoitus oli nimenomaan kannustaa ohjelmaa linnoittamaan. Kuninkaan turvallisuus määritetään siten, että bonusta saa kuninkaan edessä olevasta sotilaasta ja kahdesta etuviistossa olevasta sotilaasta. Koska pelin alussa kuninkaan edestä on yleensä siirtynyt sotilas, niin ohjelma tajuaa, että linnoittuminen on edullista, koska sen jälkeen kuninkaan edessä on kolme sotilasta suojakilpenä.

Painostus vastustajan kuningasta kohtaan on toteutettu samalla periaatteella kuin luvussa 3.4, mutta nyt huomioon otetaan vain uhkaukset yhden ruudun sisällä kuninkaasta.

Sotilasasemassa otetaan huomioon vain kaksoissotilas. Koska ohjelma vaihtoi hyvin mielellään lähetin ratsuun, pienensin kaksoissotilaan painokerrointa ja muutin lähetin arvoksi 3,25, jolloin ohjelma ei tee vaihtoa ainakaan silloin, jos ratsu on edullisesti kiinnitetty.

5.2.7 Kokonaisuus

Koko ohjelman toiminta selviää parhaiten seuraavasta kuvasta. Katso kuva 6.



Kuva 6. Ohjelman rakenne. Nuolet kuvaavat aliohjelmakutsuja.

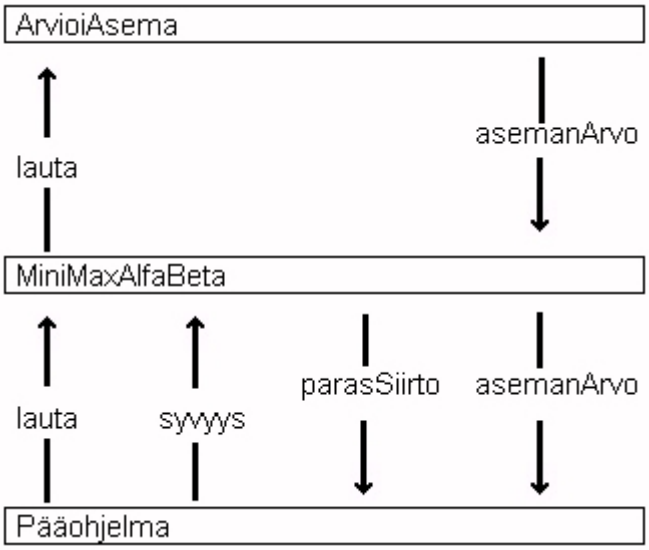
Pelin alussa ohjelma kysyy pelaajalta vaikeustason. Vaikeustaso vaikuttaa peliin siten, että ohjelma laskee pelipuuta sitä syvemmälle mitä vaikeampi vaikeustaso. Suurilla syvyyksillä ohjelma pelaa paremmin, mutta ohjelma myös joutuu tutkimaan suuren määrän asemia, jolloin ohjelman suoritus hidastuu nopeasti. Pelaaja vois myös valita ottaako valkeat vai mustat nappulat.

Ohjelmassa on toteutettu yksinkertainen avauskirjasto. Jos ohjelma on valkeilla, se tekee satunnaisesti joko siirron 1.e4, 1.d4 tai 1.c4. Jos ohjelma on mustilla, ja pelaaja pelaa 1.e4, ohjelma vastaa joko 1. - e5, 1. - e6 tai 1. - c5.

Seuraavaksi ohjelma tulostaa laudan ja kysyy pelaajan siirtoa. Kun pelaaja on syöttänyt siirtonsa, se muutetaan ohjelman tunnistamaan siirtoformaattiin. Sitten ohjelma tuottaa aseman kaikki lailliset

siirrot ja tarkistaa oliko pelaajan siirto niiden joukossa. Jos ei ollut, ohjelma ilmoittaa, että siirto oli laiton ja pyytää käyttäjää syöttämään siirron uudestaan. Kun siirto on hyväksytty, ohjelma suorittaa siirron ja tulostaa taas laudan. Pelaajan siirtovuoron lopuksi ohjelma takistaa vielä, onko asema tasapeli tai matti.

Tämän jälkeen tulee ohjelman siirtovuoro. Pääohjelma kutsuu MiniMaxAlfaBeta-hakualgoritmia. Aluksi algoritmi tuottaa kaikki lailliset siirrot ja alkaa käydä niitä läpi kuten luvussa 5.2.3 kuvattiin. Arvotusfunktio tuottaa lailliset siirrot kaksi kertaa kutakin asemaa kohti, jolloin saadaan laskettua kummankin osapuolen liikkuvuus. Jos jomman kumman liikkuvuus on nolla, tarkistetaan onko tilanne mahdollisesti matti tai patti. Jos on, ArvioiAsema-aliohjelma palauttaa tämän, muutoin se laskee muut arvotusfunktion osat ja palauttaa aseman arvon hakualgoritmille, joka etsii parhaimman mahdollisen aseman arvon. Hakualgoritmi palauttaa puolestaan pääohjelmalle tämän parhaimman arvon sekä parhaimman siirron. Katso kuva 7, josta aliohjelmien syötteet ja palautusarvot näkyvät selkeästi. Ohjelma tekee hakualgoritmin löytämän siirron, tulostaa laudan ja tarkistaa tilanteen. Jos asema ei ole tasapeli eikä matti, on jälleen pelaajan vuoro ja näin vuoronperään jatketaan kunnes TarkistaTilanne-aliohjelma havaitsee matin tai tasapelin. Pelaaja voi toki halutessaan keskeyttää pelin, joka on tässä toteutettu siten, että pelaaja antaa lähtöruuduksi nollan.



Kuva 7. Nuoli ylöspäin on syöte ja nuoli alaspäin on palautusarvo.

5.3 Testaus

Testasin ohjelmaa monilla eri tavoilla. Tässä alaluvussa on kerrottu tuloksista. Testikoneen prosessori oli viisi vuotta vanha AMD Athlon 600 MHz, joten tulokset olisivat parempia uudemmissa kotitietokoneissa.

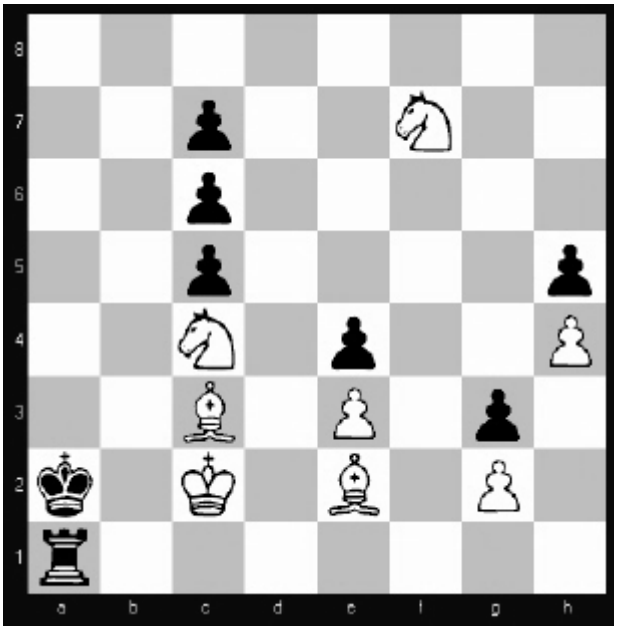
5.3.1 Pelaaminen

Haastavana tavoitteena on ollut, että ohjelman pitäisi voittaa tekijänsä shakissa. En ole kovin paljon ehtinyt pelaamaan ohjelmaa vastaan, mutta yhdessä pelissä se voitti, koska tein karkean virheen ja ohjelma pääsi voittamaan ratsuhaarukalla kuningattaren. Ohjelma, joka oli pelannut valkeilla, oli kuitenkin pelannut järkeviä kehityssiirtoja ja pisteenä i:n päälle teki nerokkaan linnoituksen kuningatarpuolelle, kun olin yrittänyt saada vastapeliä syömällä tornilla b2-sotilaan. Ohjelman pelitaso näyttäisi siis olevan tyydyttävän hyvä.

5.3.2 Oman mattitehtävän ratkaisuttaminen

Testasin syvemmin ohjelman toimintaa ja tehokkuutta yrittämällä ratkaista sen avulla itse tekemääni erittäin vaikeaa seitsemän siirron mattitehtävää, joka on julkaistu valtakunnallisessa Suomen Shakki-lehdessä [Seppälä, 2004]. Katso kuvio 19. Haastan lukijan yrittämään sen ratkaisemista. Tätä kirjoittaessani tietooni ei ole tullut vielä ketään, joka olisi pystynyt sen ratkaisemaan. Tehtävän ratkaisut ja taustat ovat liitteessä 2.

Laatija: Aki Seppälä



Kuvio 19. Seitsemän siirron matti.

7X 8+8

Aloitin mattitilanteesta ja peruutin siitä taaksepäin, ja tutkin kuinka monta asemaa ohjelma tutkii ja kuinka kauan siihen menee.

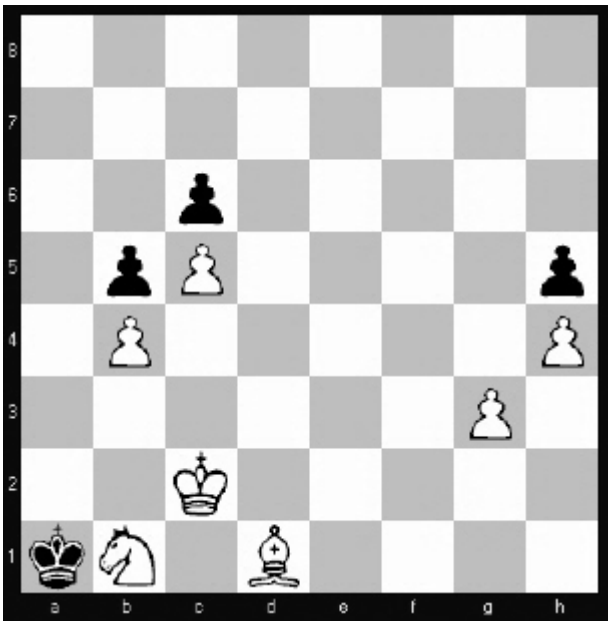
Katso taulukko 2.

Matti	Syvyys puolisiirroissa	Ratkaisuaika	Tutkitut asemat
1 X	1	alle sek	17
2 X	3	alle sek	651
3 X	5	37 sek	11039
4 X	7	14 min	223265

Taulukko 2

Keksin vielä optimoida hakua siten, että arvotusfunktio huomioi vain matin ja patin ja palautti muutoin nollan ja hakurajat (alfa ja beta) olivat 999 ja 1001, jolloin algoritmille kelpasi vain matin arvo 1000. Tutkittavien asemien määrä väheni vain parilla tuhannelle ja aikaa meni silti 12 minuuttia 25 sekuntia, joten oli selvää, että ohjelma ei selviytyisi 7X matista säädyllisessä ajassa. Ohjelma kävi läpi vain 300 asemaa sekunnissa, kun esimerkiksi Colossus Chess on noin tuhat kertaa nopeampi, mutta siltäkin menee ratkaisun löytämiseen puoli tuntia ja muiden mattimahdollisuuksien etsimisen kanssa peräti 50 minuuttia. Arvioisin, että ohjelma saattaisi selvittää tämän seitsemän siirron matin parhaimmillaan vasta parissa viikossa, joten siirryin tehtävän aiempaan versioon, joka oli kuuden siirron matti. Katso kuvio 20.

Laatija: Aki Seppälä



Kuvio 20. Kuuden siirron matti.

6X 7+4

Tämän tehtäväversion ohjelma pystyi ratkaisemaan kokonaan. Katso taulukko 3.

Matti	Syvyys puolisiirroissa	Ratkaisuaika	Tutkitut asemat
4X	7	1 min 50 sek	38241
5X	9	30 min 15 sek	659041
6X	11	8 t 5 min 40 sek	8784304

Taulukko 3.

Ohjelma ei siis seonnut monenkaan tunnin miettimisestä, vaan löysi ainoan oikean ratkaisun täsmällisesti. Aikaa tähän tosin meni todella paljon.

5.3.3 Peliaseman analysointi

Tiesin, että pullonkaula on laillisten siirtojen tuottaminen, joten testasin vielä ohjelmaa vertailemalla mahdollisten siirtojen ja laillisten siirtojen tuottamisen eroa ja myös alfa-beta-karsinnan vaikutusta. Mattitehtävät eivät tähän testiin sopineet, koska ohjelmani ei ainakaan vielä osaa tunnistaa mattia luotettavasti mahdollisten siirtojen tuottamisen avulla, joten otin lähtöasemaksi luvun 4 kuvion 13 aseman. Tässä hakusyvyys on siis kaksi siirtoa eli neljä puolisiirtoa. Katso taulukko 4.

Hakualgoritmi	Arvotusfunktio	Alfa-beta	Tutkitut asemat	Ratkaisuaika
Mahdolliset siirrot	Lailliset siirrot	käytössä	80384	52 min 30 sek
Lailliset siirrot	Lailliset siirrot	käytössä	56463	40 min 17 sek
Lailliset siirrot	Mahdolliset siirrot	ei käytössä	1828068	20 min 30 sek
Mahdolliset siirrot	Mahdolliset siirrot	ei käytössä	2116619	3 min 48 sek
Lailliset siirrot	Mahdolliset siirrot	käytössä	42012	1 min 55 sek
Mahdolliset siirrot	Pelkkä materiaali	ei käytössä	2116619	33 sek
Mahdolliset siirrot	Mahdolliset siirrot	käytössä	118940	14 sek
Mahdolliset siirrot	Pelkkä materiaali	käytössä	78052	alle 2 sek

Taulukko 4.

Kaikki ohjelmaversiot löysivät voittosiirron Da4. Loppuaseman arvo oli 2, jos arvotusfunktio otti huomioon vain materiaalin, 2,39, jos arvotusfunktio laski liikkuvuuden mahdollisista siirroista ja

2,33, jos arvotusfunktio laski liikkuvuuden laillisista siirroista. Alfa-beta-karsinta pienensi tutkittavien asemien määrää noin 95 % ja saman verran pieneni myös ratkaisuaika. Laillisten siirtojen käyttäminen pidensi ratkaisuaikaa moninkertaisesti sekä hakualgoritmissa että arvotusfunktiossa, mutta erityisesti arvotusfunktiossa.

5.4.4 Oman uuden lineaarisen hakustrategian testaus

Lopuksi testasin omaa radikaalia hakustrategiaani, joka on esitetty luvussa 4.4.3. Testiasema oli kuvion 18 asema, jota siis on pidetty shakkiohjelmille täysin mahdottomana ratkaistavana. Haku toimi siten, että tuotettiin valkean kaikki mahdolliset siirrot ja valittiin näistä siirroista paras siten, että jokainen siirto laskettiin kahden puolisiirron syvyyteen, valittiin paras ensimmäinen siirto, tehtiin siirto, laskettiin tästä asemasta taas kahden puolisiirron syvyyteen, valittiin paras siirto, tehtiin siirto ja niin edelleen, kunnes oltiin saavutettu varsinainen hakusyvyys. Kun hakusyvyys oli alle 100 puolisiirtoa, ohjelma siirsi aina 1.bxa5. Mutta kun asetin hakusyvyydeksi 102 puolisiirtoa, ohjelma siirsi 1.Kc6!. Ohjelma pystyi laskemaan alle sekunnissa 50 siirron päähän ja totemaan, että jos valkea ei syö tornia, peli on tasapeli. Vertasin tuloksia Deep Blue-ohjelmaan, joka käy läpi noin 200 miljoonaa asemaa sekunnissa. Valkealla on asemassa 6 laillista siirtoa ja mustalla 17 laillista siirtoa, joten keskimäärin kummallakin on 12 laillista siirtoa. Tasapelin toteamiseksi pitää laskea sadan puolisiirron päähän, joten asemia on käytävä läpi $12^{100} = 8 \times 10^{107}$. Kun tämä jaetaan 200 miljoonalla, 60:lla (saadaan minuutit), 60:lla (saadaan tunnit), 24:lla (saadaan päivät) ja 365:lla, saadaan vuosina Deep Bluen vaatimana suoritusaika, joka olisi $1{,}3 \times 10^{92}$. Vaikka ohjelma käyttäisi, kuten käyttääkin, alfa-beta-karsintaa, lukema olisi edelleen tähtitieteellisen suuri. Olen koonnut tulokset taulukkoon 5. Katso taulukko 5.

Ohjelma	Ratkaisuaika	Asemia tutkittu	Alfa-beta
Perusshakki	alle sekunti	21875	käytössä
Perusshakki	2 sekuntia	101808	ei
Deep Blue	yli kvindekiljoona (10^{90}) vuotta	$8{,}3 \times 10^{107}$	ei

Tauluko 5. Ohjelmien ratkaisuaikojen vertailu kuvion 18 asemassa.

5.3.5 Johtopäätökset

Testit osoittivat, että ohjelma toimii moitteettomasti kaikissa syvyyksissä. Samalla tuli todistettua

alfa-beta-karsinnan suuri teho, vaikka voittosiirto olisi lähempänä siirtolistan loppua kuin alkua. Lisäksi tuli osoitettua, että Shannonin idea laillisten siirtojen tuottamiseen periaatteessa toimii, mutta hidastaa samalla valtavasti ohjelman suoritusta. Lopuksi osoitettiin, että kehittämäni uusi lineaarinen hakustrategia pienentää todella merkittävästi haettavien asemien määrää ja ratkaisuaikaa.

5.4 Parannusideat

Ohjelman vahvin lenkki on hakualgoritmi, joka esimerkiksi mattitehtävissä käy läpi asemia suunnilleen saman verran kuin Colossus Chess X. Pelitilanteita varten hakualgoritmia voisi toki edelleen parantaa hieman lisäämällä iteratiivinen syvennys sekä best first ja killer-heuristiikka menetelmät kuten luvussa 4 on esitetty.

Ohjelman heikoin lenkki on siirtojen tuottaminen. Käyttämäni lukujen 2.1 ja 2.2 välimuodon menetelmän sijasta tehokkaampaa olisi käyttää puhdasta luvun 2.2 menetelmää tai vieläkin mieluummin luvun 2.3 tai 2.4 menetelmää. Lisäksi laillisten siirtojen tuottamisesta pitäisi luopua kokonaan. Suurin osa mahdollisista siirroista on myös laillisia, joten jokaisen mahdollisen siirron laillisuuden tarkistamiseen menee paljon aikaa. Tehokkaampi menetelmä olisi käyttää mahdollisten siirtojen listaa ja huomioida siirtojen laillisuus vasta hakupuuta rakennettaessa, jolloin laiton siirto havaitaan siitä, että kuningas on syötävissä [Gillogly, 1972]. Tätä menetelmää käytti myös CHESS 4.5-ohjelma [Frey, 1978], ja luultavasti samaa menetelmää käyttävät myös kaikki nykyiset huippuohjelmat. Siirtojen tuottamiseen menee joka tapauksessa eniten aikaa, noin 77 % ohjelman suoritusajasta [Gillogly, 1972], joten hidas siirtojen tuottaminen puurouttaa helposti hakualgoritmin, vaikka hakualgoritmi sinällään olisikin optimaalinen.

Ohjelman arvotusfunktio on tyydyttävä. Se on melko yksinkertainen, mutta samalla myös nopea. Arvotusfunktio myös toimii käytännössä melko hyvin. Olen havainnut vain pieniä puutteita. Kehitysfunktioon voisi ottaa mukaan myös linnoittumisen, jolloin ohjelma linnoittaisi varmemmin kuin nyt. Lisäksi oikean keskustakaavion lisääminen olisi suotavaa, jotta ohjelma tietäisi, että esimerkiksi Rf3 on parempi siirto kuin Re2.

Ohjelma hallitsee shakin säännöt lähes täydellisesti. Ohjelma osaa tuottaa kaikki mahdolliset siirrot ohestalyöntiä lukuun ottamatta, mutta hallitsee pienkorotukset ja linnoittumisen. Etenkin pienkorotuksia ei ole huomioitu kaikissa shakkiohjelmissa. Tähän ohjelmaan ei ole lisätty

ohestalyöntiä, koska sen toteuttaminen olisi monimutkaista ja tätä erikoissiirtoa tarvitaan hyvin harvoin. Sen toteuttaminen vaatii, että pidetään kirjaa vastustajan edellisestä siirrosta, koska ohestalyönti on mahdollinen vain, jos vastustajan sotilas on edellisellä siirrolla suorittanut sotilaan kaksoisaskeleen. Lisäksi siirron peruuttaminen olisi hankalaa, koska siirrosta olisi vaikea päätellä, onko se normaali syönti vai ohestalyönti ja lisäksi ei riitä alkunappula ja kohdenappula vaan tarvitsee päivittää myös viereinen nappula. Tähän tarvittaisiin melkein erityinen ohestalyöntilippu, joka kertoo, milloin ohestalyönti on tapahtunut. Ohestalyöntimahdollisuus olisi toki hyvä lisätä myöhemmin. Ainut toinen asia, mitä ohjelma ei hallitse, on saman aseman toistuminen kolme kertaa saman pelin aikana. Tämän toiminnon voisi toteuttaa esimerkiksi niin, että laitetaan kaikki pelatut asemat listaan ja verrataan, onko siellä kolme samanlaista asemaa. Kevyempi ratkaisu olisi säilyttää taulukossa kolmea edellistä siirtoparia, jolloin havaittaisiin kolme samaa peräkkäistä siirtoa, muttei välttämättä kolmea samaa asemaa pelissä.

Graafinen käyttöliittymä ja kaikenlaiset muut shakkiohjelmien perustoiminnot kuten käyttäjän mahdollisuus asetella nappuloita laudalle ja analysoida asemia olisivat toki hienoja, mutta ne eivät enää kuulu tämän tutkielman piiriin. Tarkoituksena oli ohjelmoida tekoäly, ja se tehtävä on nyt suoritettu.

6 YHTEENVETO

Tutkielmassa selvitettiin kuinka shakkiohjelmalle luodaan tekoäly. Shakkitekoäly koostuu kolmesta osasta, ja nämä käsiteltiin kukin omassa luvussaan. Luvussa 2 käsiteltiin sallittujen siirtojen tuottamista, luvussa 3 aseman arviointia ja luvussa 4 hakutekniikoita. Luvussa 5 sovellettiin näitä tekniikoita käytännössä ohjelmoimalla oma esimerkkisovellus. Sovelluksen testauksessa käytettiin ohjelmaa vastaan pelaamista, peliasemaa yhdestä avausansasta ja tekijän itse laatimaa mattitehtävää.

Tarkoituksena oli tutkia jo olemassa olevia tekniikoita, mutta sivutuotteena syntyi jonkin verran uuttakin. Uusia asioita olivat kehittämäni uudenlainen keskustakaavio (luku 3.6.3) ja luvun 4.4.3 ideat horisonttiongelman ratkaisemiseksi hakustrategiaa muuttamalla. Mullistavin idea näistä on *lineaarinen hakustrategia*, jossa haun aikana tasaisin välein katkaistaan koko hakupuu ja muodostetaan uusi hakupuu, jolloin siis saavutetaan lineaarinen hakuaika. Vaikka haku ei olekaan täydellinen, se on silti hyvin huomionarvoinen menetelmä ja hyvä jatkotutkimusaihe, koska kaikki aiemmat hakumenetelmät ovat olleet eksponentiaalisesti kasvavia. Luvussa 5.3.5 havaittiin, että tällä uudella hakumenetelmällä tavallinen kotitietokone voisi ratkaista jopa alle sekunnissa mahdottomana pidetyn tehtävän. Tämän tyyppisten tehtävien ratkaisemiseen perinteisellä haulla varustetulta maailman tehokkaimmalta supertietokoneelta menisi arviolta noin kvindekiljoona vuotta, joka on suuruusluokkaa maailmankaikkeuden ikä potenssiin yhdeksän! Lisäksi sovellusvaiheessa syntyi yhdistelmä lukujen 2.1 ja 2.2 menetelmistä siirtojen tuottamiseen ja useita yksinkertaisempia tapoja toteuttaa luvun 3 arvotusfunktion osia.

Tutkimus on saavuttanut tavoitteensa tai mahdollisesti ylittänyt ne. Omaksuin shakkitekniikat ja pystyin luomaan myös omia teorioita ja ohjelmoimaan reilussa viikossa ohjelman, joka pystyi voittamaan tekijänsä. Tutkimuksen alussa tämä tuntui hyvin haastavalta, ja ohjelmointiin arveltiin kuluvan useampi kuukausi.

Tutkimuskysymyksiin mitä eri shakkitekniikoita on olemassa ja miten käytännössä ohjelmoidaan toimiva kokonaisuus on siis saatu vastaukset. Olen saavuttanut omat henkilökohtaiset tavoitteeni, mutta myös lukija voi hyötyä tämän tutkielman annista monin tavoin. Suurimman hyödyn saavat oman shakkiohjelman tekemistä suunnittelevat, koska tutkielma yhdistää yleistajuisesti teorian ja käytännön. Lisäksi tämä sopii shakista kiinnostuneille, joita kiinnostaa kuinka shakkiohjelma pääpiirteissään toimii ja ohjelmoinnista kiinnostuneet voivat soveltaa näitä ideoita muihinkin

peleihin. Tämä tutkimus kohdistuu normaalishakkiin, mutta tuloksia voi myös soveltaa shakkipelin muihinkin variaatioihin, kuten Bobby Fischerin kehittämään Fischer Random shakkiin, jossa nappuloiden alkupaikat arvotaan tai Jorma Kypön kehittämään kolmen pelaajan kolmishakkiin [Kankaanranta ym, 2004, s.319-334]. Nämä olisivat myös hyviä jatkotutkimusaiheita. Lisäksi tekoälyä tarvitaan monissa muissakin peleissä kuin shakkipeleissä, ja tästä tutkimuksesta on varmasti saatavissa ideoita sellaistenkin pelien tekoälyn ohjelmointiin.

LÄHTEET

Baxter Jonathan, Tridgell Andrew ja Weaver Lex, “Learning to play chess using temporal differences”, MACH LEARN. Vol. 40, no. 3, s. 243-263, 2000

Beal, Don F., “Advances in Computer Chess 4”, Pergamon Press, Englanti, 1986

Boule Marc ja Zilic Zeljko, “An FPGA Based Move Generator for the Game of Chess”, IEEE, s.71 - 74, 2002

Bryant Martin P., “Colossus Chess X Manual”, CDS Software, Englanti, 1989

Campbell Murray, Hoane Joseph ja Hsu Feng-hsiung, “Deep Blue”, Artificial Intelligence Volume 134, s. 57-83, 2002

Charmiak Eugene ja McDermott Drew, “Introduction to Artficial Intelligence”, Addison-Wesley, USA, 1985

Eppstein David, “Board Representations”, saatavilla HMTL-muodossa < URL: http://www.ics.uci.edu/~eppstein/180a/970408.html >, tarkistettu 21.12.2004

Frey Peter, “Chess skill in man and machine”, Springer-Verlag, New York, 1978

Furnkranz Johannes, “Machine Learning in Games: A Survey”, saatavilla PDF-muodossa < URL: http://www.oefai.at/cgi-bin/get-tr?paper=oefai-tr-2000-31.pdf >, tarkistettu 21.12.2004

Gillogly James J., “The Technology Chess Program”, Artificial Intelligence Vol. 3, s. 145-163, 1972

Hietanen Päivi, ”C++ ja olio-ohjelmointi”, Teknolit Oy, Jyväskylä, 1996

Hyvönen Eero, Kararanta Ilkka ja Syrjänen Markku, “Tekoälyn ensyklopedia”, Gaudeamus, Hämeenlinna, 1993

Innala Heikki-Pekka, “Oppivat ohjelmat”, Suomen Shakki, Numero 3, s.95-97, 2001

Innala Heikki-Pekka, “Ei mitään uutta auringon alla: Kasparov – X3D Fritz 2-2”, Suomen Shakki, Numero 1, s.3-6, 2004

Kankaanranta Marja, Neittaanmäki Pekka ja Häkkinen Päivi, “Digitaalisten pelien maailmoja”, Agora Center, Jyväskylä, 2004

Konsala Kimmo, “Kaksi shakkineroa”, Karisto, Hämeenlinna, 1991

Kozdrowicki Edward W. ja Cooper Dennis W., “COKO III: the Cooper-Koz chess program”, Communications of the ACM Vol.16 n.7, s.411-427, 1973

Kotok Alan, “A Chess Playing Program for the IBM 7090 Computer”, Bachelor of Science, 1962

Levy David, “The Chess Computer Handbook”, Batsford, Lontoo, 1984

Levy David, “Computer Chess Compendium”, Batsford, Lontoo, 1988

Moreland Bruce, “0x88 Move Generation”, saatavilla HTM-muodossa < URL: http://www.seanet.com/~brucemo/topics/0x88.htm >, tarkistettu 21.12.2004

Newell Allen, Shaw Cliff ja Simon Herbert, ”Chess Playing Programs and the Problem of Complexity”, Journal of Research and Development, Numero 2, s.320-335, 1958

Norvig Peter, “Paradigms of Artificial Intelligence programming: Case Studies in Common Lisp”, Morgan Kaufmann Publishers, San Mateo, 1992

Nunn John, Burgess Graham, Emms John ja Gallagher Joe, “Nunn's Chess Openings”, Gambit, Lontoo, 1999

Samuel Arthur L., “Some Studies in Machine Learning Using the Game of Checkers”, IBM Journal, s. 211-229, 1959

Schaeffer Jonathan ja Plaat Aske, “New advances in Alpha-Beta searching”, ACM, s.124 - 130, 1996

Seppälä Aki, “Suomen Shakin ensipainoksia”, tehtävä numero 3899, Suomen Shakki, s.325, Numero 11, 2004

Shannon Claude E., “Programming a computer for playing chess”, Philosophical Magazine Vol. 41, s.256-275, 1950

Slater Eliot, “Statistics for the Chess Computer and the Factor of Mobility”, Proceedings of the Symposium on Information Theory, s. 150-152, 1950

Winston Patrick H. ja Brown Richard H., “Artificial intelligence: an mit perspective”, MIT Press, USA, 1979

LIITTEET

Liite 1. Shakkiohjelman koodi.

```
/* ************************************************************
SHAKKI.C          Versio 1.0
Aki Seppälä       20.12.2004

Ohjelman kuvaus:
Toteutetaan perustekniikoilla shakkia pelaava ohjelma.

Toteutus:

Laudan representaatio on 12x12 taulukko, jossa kaksi ulointa riviä
auttavat siirtojen tuottamisessa, koska havaitaan milloin nappula
on ulkona oikealta laudalta. Oikean laudan ulkopuolella oleva ruudun arvo
on 99. 12x10 taulukko ei riitä tässä, koska taulukkoa käsitellään indekseillä
eikä yksittäisillä numeroilla. Nappulat esitetään numeroina Shannonin
ehdotuksen mukaisesti eli valkealla s=1, R=2, L=3, T=4, D=5, K=6 ja
mustalla vastaavat arvot miinusmerkillä varustettuna s=-1, R=-2, L=-3,
T=-4, D=-5 ja K=-6.

Siirrot tuotetaan matemaattisilla laskutoimituksilla suoraan
nappuloiden liikkumissääntöihin ja taulukon indekseihin
perustuen.

Kukin asema arvioidaan arvotusfunktiolla ottamalla huomioon seuraavat
osa-alueet siten että jos valkea ja musta ovat tasapainossa
osa-alueessa aseman arvo ei muutu, mutta aina
jos valkealla on etua osa-alueessa aseman arvo kasvaa ja vastaavasti jos
mustalla on etua aseman arvo pienenee:

- materiaali (K=1000, D=9, T=5, L=3,25, R=3, s=1)
- liikkuvuus (aseman lailliset siirrot kertoimella 0,03)
- keskustan hallinta (hyvin yksinkertainen tapa, annetaan hieman bonusta
  jokaisesta nappulasta, joka valtaa ruudun kekustaneliöstä e5-d5-d4-e4)
- kehitys: kehitysfunktiona kevyet upseerit pois paikoiltaan / 3
  ja lisäksi rangaistaan liian aikaisesta daamin liikkumisesta
- sotilasasema (kaksoissotilaista sakotetaan)
- kuningashyökkäys (uhkauksista vastustajan kuninkaan viereen saa bonusta)
- kuninkaan turvallisuus (annetaan bonusta suoraan oman kuninkaan edessä
  olevista sotilaista)

Haku suoritetaan perus minimax alfa betalla, joka etsii asemasta
molemminpuolin parhaan jatkon vuoron perään minimoimalla ja maksimoimalla
käyttäen alfa-beta-karsintaa, joka karsii suuren osan epäoleellisista
asemista pois.


************************************************************** */
#include<stdio.h>
#include<iostream.h>
#include<stdlib.h>
#include<time.h>

struct Siirto
{
```

```
  int alkuRivi;
  int alkuSarake;
  int kohdeRivi;
  int kohdeSarake;
  int korotusNappula;
  Siirto *seuraava;
};

long int asemaLaskuri=0;
int siirrotIlmanSotilaidenSiirtoja=0;

TeeSiirto(int, int, int, int, int, int[12][12]);
int KumoaSiirto(int, int, int, int, int, int, int[12][12]);
double ArvioiAsema(int [12][12],int [4], int);
double LaskeMateriaali(int [12][12]);

int LisaaSiirto(Siirto *&p_alku, int p_alkuRivi, int p_alkuSarake, int
p_kohdeRivi, int p_kohdeSarake, int p_korotusNappula)

{
  Siirto *uusi = new Siirto;
  if (uusi)
  {
    uusi->alkuRivi = p_alkuRivi;
    uusi->alkuSarake = p_alkuSarake;
    uusi->kohdeRivi = p_kohdeRivi;
    uusi->kohdeSarake = p_kohdeSarake;
    uusi->korotusNappula = p_korotusNappula;
    uusi->seuraava = NULL;
    Siirto *kohdalla;
    if (!p_alku)
      p_alku = uusi;
    else
    {
      kohdalla = p_alku;
      while (kohdalla->seuraava)
        kohdalla = kohdalla->seuraava;
      kohdalla->seuraava = uusi;
    }  {
    return 1;}
  }
  else {
    cout << "Listaan lisäys epäonistui!!";
    return 0; }
}

int TyhjennaLista(Siirto *&alku)
{
  Siirto *seuraava;
  Siirto *kohdalla = alku;

  while (kohdalla)
  {
    seuraava = kohdalla->seuraava;
    delete kohdalla;
    kohdalla = seuraava;
  }
  *&alku = NULL;

  return 0;
}

int TuotaSotilaanSiirrot(int lauta[12][12], int siirtovuoro,int rivi, int
sarake,Siirto *&alku)
```

```
{

    // tuotetaan sotilaan yksi askel eteenpäin jos edessä oleva ruutu tyhjä
    // ehdolla että sotilas ei ole vielä korottumassa
    if (lauta[rivi-(1*siirtovuoro)][sarake] == 0 && ((rivi > 3 &&
siirtovuoro==1) || (rivi < 8 && siirtovuoro==-1))) {

    LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake,0);
  }

  // jos valkea sotilas on rivillä 3 tai musta sotilas rivillä 8
    // tapahtuu nappulan korotus joko ratsuksi, lähetiksi, torniksi
    // tai kuningattareksi ja kaikki nämä siirtovaihtoehdot tuotetaan
    if(lauta[rivi-(1*siirtovuoro)][sarake] == 0 && rivi==5.5-siirtovuoro*2.5){
      // korotetaan sotilaasta daami
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake,5*siirtovuoro);
      // korotetaan sotilaasta torni
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake,4*siirtovuoro);
      // korotetaan sotilaasta lähetti
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake,3*siirtovuoro);
      // koretetaan sotilaasta ratsu
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake,2*siirtovuoro);
    }

  // kaksoisaskel mahdollinen vain jos valkea sotilas rivillä 8
  // tai musta sotilas rivillä 3
  if (rivi==5.5+siirtovuoro*2.5 && lauta[rivi-(1*siirtovuoro)][sarake] == 0 &&
lauta[rivi-(2*siirtovuoro)][sarake] == 0)
  {
    LisaaSiirto(alku,rivi,sarake,rivi-(2*siirtovuoro),sarake,0);
  }

  // sotilas syö vasemmalle
  if ((lauta[rivi-(1*siirtovuoro)][sarake-1] < 0 && siirtovuoro==1 && rivi > 3)
|| (lauta[rivi-(1*siirtovuoro)][sarake-1] > 0 && siirtovuoro==-1  && lauta[rivi-
(1*siirtovuoro)][sarake-1] != 99 && rivi < 8))
  {
    LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake-1,0);
    }

    if(rivi==5.5-siirtovuoro*2.5 && ((lauta[rivi-(1*siirtovuoro)][sarake-1] < 0
&& siirtovuoro==1) || (lauta[rivi-(1*siirtovuoro)][sarake-1] > 0 &&
siirtovuoro==-1  && lauta[rivi-(1*siirtovuoro)][sarake-1] != 99)))

    {
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake-1,5*siirtovuoro);
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake-1,4*siirtovuoro);
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake-1,3*siirtovuoro);
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake-1,2*siirtovuoro);
    }
  // sotilas syö oikealle
  if ((lauta[rivi-(1*siirtovuoro)][sarake+1] < 0 && siirtovuoro==1 && rivi > 3)
|| (lauta[rivi-(1*siirtovuoro)][sarake+1] > 0 && siirtovuoro==-1 && lauta[rivi-
(1*siirtovuoro)][sarake+1] != 99 && rivi < 8))
  {
    LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake+1,0);
    }
    if(rivi==5.5-siirtovuoro*2.5 && ((lauta[rivi-(1*siirtovuoro)][sarake+1] < 0
&& siirtovuoro==1) || (lauta[rivi-(1*siirtovuoro)][sarake+1] > 0 &&
siirtovuoro==-1 && lauta[rivi-(1*siirtovuoro)][sarake+1] != 99)))
    {
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake+1,5*siirtovuoro);
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake+1,4*siirtovuoro);
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake+1,3*siirtovuoro);
```

```
      LisaaSiirto(alku,rivi,sarake,rivi-(1*siirtovuoro),sarake+1,2*siirtovuoro);
    }
return 0;
}

int TuotaRatsunSiirrot(int lauta[12][12], int siirtovuoro, int rivi, int sarake,
Siirto *&alku)
{
  if (lauta[rivi-2][sarake-1] != 99  && ((lauta[rivi-2][sarake-1] < 1  &&
siirtovuoro==1) || (lauta[rivi-2][sarake-1] > -1  && siirtovuoro==-1))) {
    LisaaSiirto(alku,rivi,sarake,rivi-2,sarake-1,0);
        }
        if (lauta[rivi-1][sarake-2] != 99 && ((lauta[rivi-1][sarake-2] < 1 &&
siirtovuoro==1) || (lauta[rivi-1][sarake-2] > -1 && siirtovuoro==-1))) {
          LisaaSiirto(alku,rivi,sarake,rivi-1,sarake-2,0);
        }
        if (lauta[rivi+1][sarake-2] != 99 && ((lauta[rivi+1][sarake-2] < 1 &&
siirtovuoro==1) || (lauta[rivi+1][sarake-2] > -1 && siirtovuoro==-1))) {

        LisaaSiirto(alku,rivi,sarake,rivi+1,sarake-2,0);
        }
        if (lauta[rivi+2][sarake-1] != 99 && ((lauta[rivi+2][sarake-1] < 1 &&
siirtovuoro==1) || (lauta[rivi+2][sarake-1] > -1 && siirtovuoro==-1))) {
          LisaaSiirto(alku,rivi,sarake,rivi+2,sarake-1,0);
        }
        if (lauta[rivi+2][sarake+1] != 99 && ((lauta[rivi+2][sarake+1] < 1 &&
siirtovuoro==1) || (lauta[rivi+2][sarake+1] >-1 && siirtovuoro==-1))) {
          LisaaSiirto(alku,rivi,sarake,rivi+2,sarake+1,0);
        }
        if (lauta[rivi+1][sarake+2] != 99 && ((lauta[rivi+1][sarake+2] < 1 &&
siirtovuoro==1) || (lauta[rivi+1][sarake+2] > -1 && siirtovuoro==-1))) {
          LisaaSiirto(alku,rivi,sarake,rivi+1,sarake+2,0);
        }
        if (lauta[rivi-1][sarake+2] != 99 && ((lauta[rivi-1][sarake+2] < 1 &&
siirtovuoro==1) || (lauta[rivi-1][sarake+2] >-1 && siirtovuoro==-1))) {
          LisaaSiirto(alku,rivi,sarake,rivi-1,sarake+2,0);
        }
        if (lauta[rivi-2][sarake+1] != 99 &&((lauta[rivi-2][sarake+1] < 1 &&
siirtovuoro==1) || (lauta[rivi-2][sarake+1] >-1 && siirtovuoro==-1))) {
          LisaaSiirto(alku,rivi,sarake,rivi-2,sarake+1,0);
        }
return 0;
}

int TuotaLahetinSiirrot(int lauta[12][12],int siirtovuoro, int rivi, int
sarake,Siirto *&alku)
{
    int syontilippu = 0;
    int i=1;
  // tutkitaan vasen yläviisto
  // lähetti voi siirtyä jos yläviistoruutu on tyhjä tai siinä on
  // vastustajan nappula
  while (lauta[rivi-i][sarake-i] != 99  && syontilippu==0 && ((lauta[rivi-
i][sarake-i] < 1  && siirtovuoro==1) || (lauta[rivi-i][sarake-i] > - 1  &&
siirtovuoro==-1)))
  {
        LisaaSiirto(alku,rivi,sarake,rivi-i,sarake-i,0);

        // jos syötiin vastustajalta nappula, lähetti ei pääse enää tästä
eteenpäin
        if((lauta[rivi-i][sarake-i] < 0 && siirtovuoro==1) || (lauta[rivi-
i][sarake-i] > 0 && siirtovuoro==-1 ))
          syontilippu = 1;
        i++;
```

```
  }
        // varmistetaan, että syöntilippu ei ole päällä
        syontilippu = 0;
        // asetaan apumuuttaja jälleen ykköseksi
        i=1;

        // nyt tutkitaan lähetin siirrot oikeaan yläviistoon

        while (lauta[rivi-i][sarake+i] != 99  && syontilippu==0 && ((lauta[rivi-
i][sarake+i] < 1  && siirtovuoro==1) || (lauta[rivi-i][sarake+i] > - 1  &&
siirtovuoro==-1)))
        {

        LisaaSiirto(alku,rivi,sarake,rivi-i,sarake+i,0);

        if((lauta[rivi-i][sarake+i] < 0 && siirtovuoro==1) || (lauta[rivi-
i][sarake+i] > 0 && siirtovuoro==-1 ))
          syontilippu = 1;
        i++;
        }

        // varmistetaan, että syöntilippu ei ole päällä
        syontilippu = 0;
        // asetaan apumuuttaja jälleen ykköseksi
        i=1;

        // nyt tutkitaan lähetin siirron oikeaan alaviistoon
        while (lauta[rivi+i][sarake+i] != 99  && syontilippu==0 &&
((lauta[rivi+i][sarake+i] < 1  && siirtovuoro==1) || (lauta[rivi+i][sarake+i] >
- 1  && siirtovuoro==-1)))
        {

        LisaaSiirto(alku,rivi,sarake,rivi+i,sarake+i,0);

        if((lauta[rivi+i][sarake+i] < 0 && siirtovuoro==1) ||
(lauta[rivi+i][sarake+i] > 0 && siirtovuoro==-1 ))
          syontilippu = 1;
        i++;
        }
        // varmistetaan, että syntilippu ei ole päällä
        syontilippu = 0;
        // asetaan apumuuttaja jälleen ykköseksi
        i=1;

        // lopuksi tutkitaan lähetin siirtomahdollisuudet vasempaan alakulmaan
        while (lauta[rivi+i][sarake-i] != 99  && syontilippu==0 &&
((lauta[rivi+i][sarake-i] < 1  && siirtovuoro==1) || (lauta[rivi+i][sarake-i] >
- 1  && siirtovuoro==-1)))
        {
          LisaaSiirto(alku,rivi,sarake,rivi+i,sarake-i,0);

        if((lauta[rivi+i][sarake-i] < 0 && siirtovuoro==1) ||
(lauta[rivi+i][sarake-i] > 0 && siirtovuoro==-1 ))
          syontilippu = 1;
        i++;
        }
return 0;
}

int TuotaTorninSiirrot(int lauta[12][12],int siirtovuoro, int rivi, int sarake,
Siirto *&alku)
{

    int syontilippu = 0;
```

```
    int i=1;

    // tutkitaan ensin tornin liikkket ylös
    while (lauta[rivi-i][sarake] != 99  && syontilippu==0 && ((lauta[rivi-
i][sarake] < 1  && siirtovuoro==1) || (lauta[rivi-i][sarake] > - 1  &&
siirtovuoro==-1)))
        {
        LisaaSiirto(alku,rivi,sarake,rivi-i,sarake,0);

        if((lauta[rivi-i][sarake] < 0 && siirtovuoro==1) || (lauta[rivi-
i][sarake] > 0 && siirtovuoro==-1 ))
          syontilippu = 1;
        i++;
        }

        // varmistetaan, että syöntilippu ei ole päällä
        syontilippu = 0;
        // asetaan apumuuttaja jälleen ykköseksi
        i=1;

        // tuotetaan tornin liikkeet alas
        while (lauta[rivi+i][sarake] != 99  && syontilippu==0 &&
((lauta[rivi+i][sarake] < 1  && siirtovuoro==1) || (lauta[rivi+i][sarake] > - 1
&& siirtovuoro==-1)))
        {
        LisaaSiirto(alku,rivi,sarake,rivi+i,sarake,0);

        if((lauta[rivi+i][sarake] < 0 && siirtovuoro==1) ||
(lauta[rivi+i][sarake] > 0 && siirtovuoro==-1 ))
          syontilippu = 1;
        i++;
        }

        // varmistetaan, että syöntilippu ei ole päällä
        syontilippu = 0;
        // asetaan apumuuttaja jälleen ykköseksi
        i=1;

        // tuotetaan tornin liikkeet vasemmalle
        while (lauta[rivi][sarake-i] != 99  && syontilippu==0 &&
((lauta[rivi][sarake-i] < 1  && siirtovuoro==1) || (lauta[rivi][sarake-i] > - 1
&& siirtovuoro==-1)))
        {
        LisaaSiirto(alku,rivi,sarake,rivi,sarake-i,0);


        if((lauta[rivi][sarake-i] < 0 && siirtovuoro==1) || (lauta[rivi][sarake-
i] > 0 && siirtovuoro==-1 ))
          syontilippu = 1;
        i++;
        }

        // varmistetaan, että syöntilippu ei ole päällä
        syontilippu = 0;
        // asetaan apumuuttaja jälleen ykköseksi
        i=1;

        // tuotetaan tornin liikkeet oikealle
        while (lauta[rivi][sarake+i] != 99  && syontilippu==0 &&
((lauta[rivi][sarake+i] < 1  && siirtovuoro==1) || (lauta[rivi][sarake+i] > - 1
&& siirtovuoro==-1)))
        {

        LisaaSiirto(alku,rivi,sarake,rivi,sarake+i,0);
```

```
        if((lauta[rivi][sarake+i] < 0 && siirtovuoro==1) ||
(lauta[rivi][sarake+i] > 0 && siirtovuoro==-1 ))
          syontilippu = 1;
        i++;
        }
return 0;
}

int TuotaKuninkaanSiirrot(int lauta[12][12],int siirtovuoro, int rivi, int
sarake, int linnoitusoikeudet[4], Siirto *&alku)
{

    // kuninkaan siirto ylös
      if(lauta[rivi-1][sarake] != 99 && ((lauta[rivi-1][sarake] < 1 &&
siirtovuoro==1) || (lauta[rivi-1][sarake] > -1 && siirtovuoro==-1)))

    {
      LisaaSiirto(alku,rivi,sarake,rivi-1,sarake,0);

    }
    // kuninkaan siirto alas
    if(lauta[rivi+1][sarake] != 99 && ((lauta[rivi+1][sarake] < 1 &&
siirtovuoro==1) || (lauta[rivi+1][sarake] > -1 && siirtovuoro==-1)))
    {
      LisaaSiirto(alku,rivi,sarake,rivi+1,sarake,0);

    }
      // kuninkaan siirto vasemmalle
    if(lauta[rivi][sarake-1] != 99 && ((lauta[rivi][sarake-1] < 1 &&
siirtovuoro==1) || (lauta[rivi][sarake-1] > -1 && siirtovuoro==-1)))
    {
      LisaaSiirto(alku,rivi,sarake,rivi,sarake-1,0);

    }
      // kuninkaan siirto oikealle
    if(lauta[rivi][sarake+1] != 99 && ((lauta[rivi][sarake+1] < 1 &&
siirtovuoro==1) || (lauta[rivi][sarake+1] > -1 && siirtovuoro==-1)))
    {
      LisaaSiirto(alku,rivi,sarake,rivi,sarake+1,0);
    }
    // kuninkaan siirto vasempaan yläviistoon
    if(lauta[rivi-1][sarake-1] != 99 && ((lauta[rivi-1][sarake-1] < 1 &&
siirtovuoro==1) || (lauta[rivi-1][sarake-1] > -1 && siirtovuoro==-1)))
    {
      LisaaSiirto(alku,rivi,sarake,rivi-1,sarake-1,0);
    }
      // kuninkaan siirto oikeaan yläviistoon
    if(lauta[rivi-1][sarake+1] != 99 && ((lauta[rivi-1][sarake+1] < 1 &&
siirtovuoro==1) || (lauta[rivi-1][sarake+1] > -1 && siirtovuoro==-1)))
    {
      LisaaSiirto(alku,rivi,sarake,rivi-1,sarake+1,0);
    }
    // kuninkaan siirto vasempaan alaviistoon
    if(lauta[rivi+1][sarake-1] != 99 && ((lauta[rivi+1][sarake-1] < 1 &&
siirtovuoro==1) || (lauta[rivi+1][sarake-1] > -1 && siirtovuoro==-1)))
    {
      LisaaSiirto(alku,rivi,sarake,rivi+1,sarake-1,0);

    }
      // kuninkaan siirto oikeaan alaviistoon
    if(lauta[rivi+1][sarake+1] != 99 && ((lauta[rivi+1][sarake+1] < 1 &&
siirtovuoro==1) || (lauta[rivi+1][sarake+1] > -1 && siirtovuoro==-1)))
    {
```

```
      LisaaSiirto(alku,rivi,sarake,rivi+1,sarake+1,0);

    }
    // tuotetaan valkean lyhyt linna
    if (linnoitusoikeudet[0]==1 && siirtovuoro==1 && rivi==9 && sarake==6 &&
lauta[rivi][sarake+1]==0 && lauta[rivi][sarake+2]==0  &&
lauta[rivi][sarake+3]==4)
      LisaaSiirto(alku,rivi,sarake,rivi,sarake+2,0);
    // tuotetaan valkean pitkä linna
    if (linnoitusoikeudet[1]==1 && siirtovuoro==1 && rivi==9 && sarake==6 &&
lauta[rivi][sarake-1]==0 && lauta[rivi][sarake-2]==0 && lauta[rivi][sarake-3]==0
&& lauta[rivi][sarake-4]==4)
      LisaaSiirto(alku,rivi,sarake,rivi,sarake-2,0);
    // tuotetaan mustan lyhyt linna
    if (linnoitusoikeudet[2]==1 && siirtovuoro==-1 && rivi==2 && sarake==6 &&
lauta[rivi][sarake+1]==0 && lauta[rivi][sarake+2]==0 && lauta[rivi][sarake+3]==-
4)
      LisaaSiirto(alku,rivi,sarake,rivi,sarake+2,0);
    // tuotetaan mustan pitkä linna
    if (linnoitusoikeudet[3]==1 && siirtovuoro==-1 && rivi==2 && sarake==6 &&
lauta[rivi][sarake-1]==0 && lauta[rivi][sarake-2]==0 && lauta[rivi][sarake-3]==0
&& lauta[rivi][sarake-4]==-4)
      LisaaSiirto(alku,rivi,sarake,rivi,sarake-2,0);
  return 0;
}

int TuotaKaikkiMahdollisetSiirrot(int lauta[12][12], int siirtovuoro, int
linnoitusoikeudet[], Siirto *&alku)
{

  int rivi, sarake;
  for (rivi = 0; rivi < 12; rivi ++)
    for (sarake=0; sarake < 12; sarake++) {
      // tutkitaan onko valkean siirtovuoro
      if (siirtovuoro==1) {

        if (lauta[rivi][sarake]==1)
         TuotaSotilaanSiirrot(lauta,siirtovuoro,rivi,sarake,alku);

        // jos ruudun arvo on 2 eli ruudussa on valkea ratsu
        if (lauta[rivi][sarake]==2)
          TuotaRatsunSiirrot(lauta,siirtovuoro,rivi,sarake,alku);
        //jos ruudun arvo on 3 eli ruudussa on valkea lähetti
        if (lauta[rivi][sarake]==3)
        TuotaLahetinSiirrot(lauta,siirtovuoro,rivi,sarake,alku);

        // jos ruudun arvo on 4 eli ruudussa on valkea torni
        if (lauta[rivi][sarake]==4)
        TuotaTorninSiirrot(lauta,siirtovuoro,rivi,sarake,alku);

        // jos ruudun arvo on 5 eli ruudussa on valkea daami, tuotetaan
        // sekä lähetin että tornin siirrot, joista yhdessä
        // muodostuu kuningattaren siirrot
        if (lauta[rivi][sarake]==5) {
        TuotaTorninSiirrot(lauta,siirtovuoro,rivi,sarake,alku);
        TuotaLahetinSiirrot(lauta,siirtovuoro,rivi,sarake,alku);
        }
        // jos ruudun arvo on 6 eli ruudussa on valkea kuningas
        if (lauta[rivi][sarake]==6)

TuotaKuninkaanSiirrot(lauta,siirtovuoro,rivi,sarake,linnoitusoikeudet,alku);
      }
        // tutkitaan onko mustan siirtovuoro
        if (siirtovuoro==-1) {
```

```
        // jos ruudun arvo on -1 eli siinä on musta sotilas
        if (lauta[rivi][sarake]==-1)
         TuotaSotilaanSiirrot(lauta,siirtovuoro,rivi,sarake,alku);

        if (lauta[rivi][sarake]==-2)
          TuotaRatsunSiirrot(lauta,siirtovuoro,rivi,sarake,alku);
          if (lauta[rivi][sarake]==-3)
        TuotaLahetinSiirrot(lauta,siirtovuoro,rivi,sarake,alku);

        if (lauta[rivi][sarake]==-4)
        TuotaTorninSiirrot(lauta,siirtovuoro,rivi,sarake,alku);

        if (lauta[rivi][sarake]==-5) {
          TuotaTorninSiirrot(lauta,siirtovuoro,rivi,sarake,alku);
          TuotaLahetinSiirrot(lauta,siirtovuoro,rivi,sarake,alku);
        }
        if (lauta[rivi][sarake]==-6)

TuotaKuninkaanSiirrot(lauta,siirtovuoro,rivi,sarake,linnoitusoikeudet,alku);
        }
  }
  return 0;
}

int TarkistaLaillisuus(int lauta[12][12],int siirtovuoro,int linnoitus[2])
{

  int muistiAlkuNappula;
  int muistiKohdeNappula;
  int linnoitusoikeudet[4] = {0,0,0,0}; // näitä ei tarvitse tutkia
  int rivi;
  int sarake;
  int kuninkaanPaikka[2] = {0,0};
  double asemanArvo;

  Siirto *vastustajanSiirtojenAlku = NULL;

  // etsitään kuninkaan paikka

  if (linnoitus[0]==1 || linnoitus[1]==1)
  {
  for (rivi = 2; rivi < 10; rivi ++)
    for (sarake = 2; sarake < 10; sarake++)
    {
    if ((lauta[rivi][sarake]==6 && siirtovuoro==1) || (lauta[rivi][sarake]==-6
&& siirtovuoro==-1))
      kuninkaanPaikka[0]=rivi;
      kuninkaanPaikka[1]=sarake;
    }
  }

  TuotaKaikkiMahdollisetSiirrot(lauta,siirtovuoro*-
1,linnoitusoikeudet,vastustajanSiirtojenAlku);

  Siirto *kohdalla;
  kohdalla = vastustajanSiirtojenAlku;
  while (kohdalla)
  {
    muistiAlkuNappula = lauta[kohdalla->alkuRivi][kohdalla->alkuSarake];
    muistiKohdeNappula = lauta[kohdalla->kohdeRivi][kohdalla->kohdeSarake];

    TeeSiirto(kohdalla->alkuRivi,kohdalla->alkuSarake,kohdalla-
>kohdeRivi,kohdalla->kohdeSarake,kohdalla->korotusNappula, lauta);
    asemanArvo = LaskeMateriaali(lauta);
```

```
    KumoaSiirto(kohdalla->alkuRivi,kohdalla->alkuSarake, kohdalla->kohdeRivi,
kohdalla->kohdeSarake, muistiAlkuNappula,muistiKohdeNappula,lauta);
    if (asemanArvo < -500 || asemanArvo > 500) {
      TyhjennaLista(vastustajanSiirtojenAlku);
      vastustajanSiirtojenAlku = NULL;
      return 0;
    }

    // tutkitaan onko lyhyt linnoitus laillinen
    if (linnoitus[0]==1) {
      // jos musta nappula uhkaa ruutuja e1 tai g1 valkean vuorolla
      // tai valkea nappula uhkaa ruutuja e8 tai f8 mustan vuorolla
      // linnoitus on laiton
      if ((kohdalla->kohdeRivi==5.5+3.5*siirtovuoro && kohdalla->kohdeSarake==6)
|| (kohdalla->kohdeRivi==5.5+3.5*siirtovuoro && kohdalla->kohdeSarake==7))
      {
      TyhjennaLista(vastustajanSiirtojenAlku);
      vastustajanSiirtojenAlku = NULL;
        return 0; }
      }
    // tutkitaan onko pitkä linnoitus laillinen
    if (linnoitus[1]==2) {
      // jos musta nappula uhkaa ruutuja e1 tai d1 tai valkean vuorolla
      // tai valkea nappula uhkaa ruutuja e8 tai f8 mustan vuorolla
      //linnoitus on laiton
    if ((kohdalla->kohdeRivi==5.5+3.5*siirtovuoro && kohdalla->kohdeSarake==6)
|| (kohdalla->kohdeRivi==5.5+3.5*siirtovuoro && kohdalla->kohdeSarake==5))
    {
    TyhjennaLista(vastustajanSiirtojenAlku);
      vastustajanSiirtojenAlku = NULL;}
        return 0; }

    kohdalla = kohdalla->seuraava;
    }
    TyhjennaLista(vastustajanSiirtojenAlku);
    vastustajanSiirtojenAlku = NULL;
    return 1;
}

int TuotaKaikkiLaillisetSiirrot(int lauta[12][12], int siirtovuoro, int
linnoitusoikeudet[], Siirto *&laillistenSiirtojenAlku)
{
  Siirto *mahdollistenSiirtojenAlku = NULL;
  int linnoitus[2] = {0,0};
  int muistiAlkuNappula=0;
  int muistiKohdeNappula=0;


TuotaKaikkiMahdollisetSiirrot(lauta,siirtovuoro,linnoitusoikeudet,mahdollistenSi
irtojenAlku);
  Siirto *kohdalla;
  kohdalla = mahdollistenSiirtojenAlku;
  while (kohdalla)
  {
  // tutkitaan onko lähtönappula kuningas

  if (lauta[kohdalla->alkuRivi][kohdalla->alkuSarake]==6 || lauta[kohdalla-
>alkuRivi][kohdalla->alkuSarake]==-6)
  {
    if (kohdalla->kohdeSarake - kohdalla->alkuSarake == 2)
      linnoitus[0] = 1; // ilmoitetaan lipulla että on tehty linna
    if (kohdalla->alkuSarake - kohdalla->kohdeSarake == 3)
      linnoitus[1] = 1; // ilmoitetaan lipulla että on tehty pitkä linna
    }
```

```
  muistiAlkuNappula = lauta[kohdalla->alkuRivi][kohdalla->alkuSarake];
  muistiKohdeNappula = lauta[kohdalla->kohdeRivi][kohdalla->kohdeSarake];


  TeeSiirto(kohdalla->alkuRivi,kohdalla->alkuSarake,kohdalla-
>kohdeRivi,kohdalla->kohdeSarake,kohdalla->korotusNappula, lauta);

  if (TarkistaLaillisuus(lauta, siirtovuoro,linnoitus)==1)
    LisaaSiirto(laillistenSiirtojenAlku,kohdalla->alkuRivi,kohdalla-
>alkuSarake,kohdalla->kohdeRivi,kohdalla->kohdeSarake,kohdalla->korotusNappula);

    KumoaSiirto(kohdalla->alkuRivi,kohdalla->alkuSarake, kohdalla->kohdeRivi,
kohdalla->kohdeSarake, muistiAlkuNappula,muistiKohdeNappula,lauta);
    linnoitus[0]=0;
    linnoitus[1]=0;
    kohdalla = kohdalla -> seuraava;
    }

  TyhjennaLista(mahdollistenSiirtojenAlku);
  mahdollistenSiirtojenAlku = NULL;
return 0;
}

int TeeSiirto(int p_alkuRivi,int p_alkuSarake,int p_kohdeRivi,int
p_kohdeSarake,int p_korotusnappula, int lauta[12][12])
{
  int nappula;
  nappula = lauta[p_alkuRivi][p_alkuSarake];

  if (p_korotusnappula == 0) {
    lauta[p_alkuRivi][p_alkuSarake] = 0;
    lauta[p_kohdeRivi][p_kohdeSarake] = nappula;

    if (nappula == 6 || nappula == -6) {
     // tehdään lyhyt linna
      if(p_kohdeSarake-p_alkuSarake == 2 && p_kohdeSarake==8) {
        lauta[p_alkuRivi][p_alkuSarake+3]=0;
        if (nappula == 6)
          lauta[p_alkuRivi][p_alkuSarake+1]=4;
        if (nappula == -6)
          lauta[p_alkuRivi][p_alkuSarake+1]=-4;
        }
     // tehdään pitkä linna
     if(p_alkuSarake - p_kohdeSarake == 2 && p_kohdeSarake==4) {
        lauta[p_alkuRivi][p_alkuSarake-4]=0;
        if (nappula ==6)
          lauta[p_alkuRivi][p_alkuSarake-1]=4;
        if (nappula ==-6)
          lauta[p_alkuRivi][p_alkuSarake-1]=-4;
          }
    }
    }
  else {
    lauta[p_alkuRivi][p_alkuSarake] = 0;
    lauta[p_kohdeRivi][p_kohdeSarake] = p_korotusnappula;
  }

  return 0;
}

int KumoaSiirto(int muistiAlkuRivi, int muistiAlkuSarake, int muistiKohdeRivi,
int muistiKohdeSarake, int muistiAlkuNappula, int muistiKohdeNappula, int
lauta[12][12])
{
```

```
  lauta[muistiAlkuRivi][muistiAlkuSarake] = muistiAlkuNappula;
  lauta[muistiKohdeRivi][muistiKohdeSarake] = muistiKohdeNappula;

  // tutkitaan onko tehty linnoitus jolloin pitää päivittää myös torni
  // tutkitaan ensin valkean kuningas
  if (muistiAlkuNappula==6) {
    // onko tehty lyhyt linna
    if(muistiAlkuSarake==6 && muistiKohdeSarake==8)
    {
      lauta[9][7]=0;
      lauta[9][9]=4;
    }
    // onko tehty pitkä linna
    if(muistiAlkuSarake==6 && muistiKohdeSarake==4)
    {
      lauta[9][5]=0;
      lauta[9][2]=4;
    }
  }
  // tutkitaan mustan kuningas
  if (muistiAlkuNappula==-6)
  {
       // onko tehty lyhyt linna
    if(muistiAlkuSarake==6 && muistiKohdeSarake==8)
    {
      lauta[2][7]=0;
      lauta[2][9]=-4;
    }
    // onko tehty pitkä linna
    if(muistiAlkuSarake==6 && muistiKohdeSarake==4)
    {
      lauta[2][5]=0;
      lauta[2][2]=-4;
    }

  }

  return 0;
}

int TarkistaOnkoMatti(int lauta[12][12])
{
  Siirto *laillistenSiirtojenAlku = NULL;
  int linnoitusoikeudet[4] = {0,0,0,0}; // linnoitusoikeuksilla ei väliä


TuotaKaikkiMahdollisetSiirrot(lauta,1,linnoitusoikeudet,laillistenSiirtojenAlku)
;

  Siirto *kohdalla;
  kohdalla = laillistenSiirtojenAlku;
  while (kohdalla)
  {
  if (lauta[kohdalla->kohdeRivi][kohdalla->kohdeSarake] == -6)
  {
  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku=NULL;
  return 1000; }
  kohdalla = kohdalla->seuraava;
  }
  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku=NULL;
  TuotaKaikkiMahdollisetSiirrot(lauta,-
  1,linnoitusoikeudet,laillistenSiirtojenAlku);
```

```
  kohdalla = laillistenSiirtojenAlku;
  while (kohdalla)
  {

  if (lauta[kohdalla->kohdeRivi][kohdalla->kohdeSarake] == 6) {
  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku=NULL;
  return -1000;
  }
  kohdalla = kohdalla->seuraava;
  }
  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku=NULL;
  return 0;
}

int TarkistaTilanne(int lauta[12][12], int siirtovuoro)
{
  int vs=1;
  int ms=1;
  int apu=0;
  int rivi=0;
  int sarake=0;
  int kevytLaskuri=0;
  int laskuri=0;
  Siirto *laillistenSiirtojenAlku = NULL;
  int linnoitusoikeudet[4] = {0,0,0,0}; // linnoitusoikeuksilla ei väliä

  TuotaKaikkiLaillisetSiirrot(lauta,-
1,linnoitusoikeudet,laillistenSiirtojenAlku);
  if (laillistenSiirtojenAlku == NULL)
    vs = 0;

  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku=NULL;

TuotaKaikkiLaillisetSiirrot(lauta,1,linnoitusoikeudet,laillistenSiirtojenAlku);

  if (laillistenSiirtojenAlku == NULL)
    ms = 0;

  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku=NULL;

  if (ms==0 || vs==0) {
    apu=TarkistaOnkoMatti(lauta);
    if (apu==-1000)
      return -1;
    if (apu==1000)
      return 1;
    if ( (apu==0 && ms==0 && siirtovuoro==-1)
      || (apu==0 && vs==0 && siirtovuoro==1))
        return 2;
    }
    for (rivi = 2; rivi < 10; rivi ++)
    for (sarake=2; sarake < 10; sarake++)
    {
    if (lauta[rivi][sarake]== 2
    || lauta[rivi][sarake] == 3
    || lauta[rivi][sarake] == -2
    || lauta[rivi][sarake] == -3)
      kevytLaskuri++;
    if (lauta[rivi][sarake]== 4
    || lauta[rivi][sarake] == 5
```

```
    || lauta[rivi][sarake] == 1
    || lauta[rivi][sarake] == -1
    || lauta[rivi][sarake] == -4
    || lauta[rivi][sarake] == -5)
      laskuri++;
    }
    if (laskuri==0 && kevytLaskuri < 2){
      return 3;
  }

  if(siirrotIlmanSotilaidenSiirtoja >= 50)
    return 4;
  return 0;
}


int TarkistaPelaajanSiirto(int pelaajanAlkuRivi,int pelaajanAlkuSarake,int
pelaajanKohdeRivi,int pelaajanKohdeSarake, Siirto *laillistenSiirtojenAlku)
{
  Siirto *kohdalla;
  kohdalla = laillistenSiirtojenAlku;
  while (kohdalla)
  {
   if (kohdalla->alkuRivi == pelaajanAlkuRivi && kohdalla->alkuSarake ==
pelaajanAlkuSarake && kohdalla->kohdeRivi == pelaajanKohdeRivi  && kohdalla-
>kohdeSarake == pelaajanKohdeSarake)
     return 1;
   kohdalla = kohdalla->seuraava;
   }
  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku = NULL;
  return 0;
}

char NappulaMuunnos(int lauta[12][12],int pelaajanKohdeRivi, int
pelaajanKohdeSarake)
{

  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == 1
  || lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == -1)
    return 's';

  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == 2
  || lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == -2) {
    return 'R';
  }
  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == 3
  || lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == -3)
  return 'L';

  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == 4
  || lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == -4)
  return 'T';

  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == 5
  || lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == -5)
  return 'D';

  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == 6
  || lauta[pelaajanKohdeRivi][pelaajanKohdeSarake] == -6)
  return 'K';

return 'E';
}
```

```
char MuunnosSarakeRivi(int sarake)
{
  if (sarake == 2)
    return 'a';
  if (sarake == 3)
    return 'b';
  if (sarake == 4)
    return 'c';
  if (sarake == 5)
    return 'd';
  if (sarake == 6)
    return 'e';
  if (sarake == 7)
    return 'f';
  if (sarake == 8)
    return 'g';
  if (sarake == 9)
    return 'h';

return 'E';
}

int MuunnosRiviSarake(int rivi)
{
  if( rivi == 9)
    return 1;
  if( rivi == 8)
    return 2;
  if( rivi == 7)
    return 3;
  if( rivi == 6)
    return 4;
  if( rivi == 5)
    return 5;
  if( rivi == 4)
    return 6;
  if( rivi == 3)
    return 7;
  if( rivi == 2)
    return 8;

return 0;
}

int Muunnos(char jono)
{
  if (jono=='a')
    return 2;
  if (jono=='b')
    return 3;
  if (jono=='c')
    return 4;
  if (jono=='d')
    return 5;
  if (jono=='e')
    return 6;
  if (jono=='f')
    return 7;
  if (jono=='g')
    return 8;
  if (jono=='h')
    return 9;
```

```
  if (jono=='1')
    return 9;
  if (jono=='2')
    return 8;
  if (jono=='3')
    return 7;
  if (jono=='4')
    return 6;
  if (jono=='5')
    return 5;
  if (jono=='6')
    return 4;
  if (jono=='7')
    return 3;
  if (jono=='8')
    return 2;

return 0;
}

int TulostaLauta(int lauta[12][12])
{
  int rivi, sarake;
  for (rivi = 2; rivi < 10; rivi ++) {
    printf("\n\n");
    for (sarake= 2; sarake < 10; sarake++)
    {

      if (lauta[rivi][sarake]==0)
          printf("00 ");
      if (lauta[rivi][sarake]==-1)
        printf("ms ");
      if (lauta[rivi][sarake]==-2)
        printf("mR ");
      if (lauta[rivi][sarake]==-3)
        printf("mL ");
      if (lauta[rivi][sarake]==-4)
        printf("mT ");
      if (lauta[rivi][sarake]==-5)
        printf("mD ");
      if (lauta[rivi][sarake]==-6)
        printf("mK ");
      if (lauta[rivi][sarake]==1)
        printf("vs ");
        if (lauta[rivi][sarake]==2)
        printf("vR ");
      if (lauta[rivi][sarake]==3)
        printf("vL ");
      if (lauta[rivi][sarake]==4)
        printf("vT ");
      if (lauta[rivi][sarake]==5)
        printf("vD ");
      if (lauta[rivi][sarake]==6)
        printf("vK ");
    }
   }
return 0;
}

// lasketaan sellaisten nappuloiden uhkausten lukumäärä
// jotka kohdistuvat vastustajan kuninkaaseen tai heti viereiseen ruutuun
int KuningasHyokkays(int lauta[12][12])
{
  int kuningashyokkays = 0;
```

```
  Siirto *mahdollistenSiirtojenAlku = NULL;
  int rivi, sarake;
  int valkeanKuninkaanPaikka[2] = {0,0};
  int mustanKuninkaanPaikka[2] = {0,0};
  int linnoitusoikeudet[4] = {0,0,0,0};

  // etsitään kuninkaiden paikat
  for (rivi = 2; rivi < 10; rivi ++)
    for (sarake = 2; sarake < 10; sarake++)
    {
      if(lauta[rivi][sarake]==-6) {
        mustanKuninkaanPaikka[0] = rivi;
        mustanKuninkaanPaikka[1] = sarake;
    }
    if(lauta[rivi][sarake]==6) {
        valkeanKuninkaanPaikka[0] = rivi;
        valkeanKuninkaanPaikka[1] = sarake;
    }
  }


TuotaKaikkiMahdollisetSiirrot(lauta,1,linnoitusoikeudet,mahdollistenSiirtojenAlk
u);
  Siirto *kohdalla;
  kohdalla = mahdollistenSiirtojenAlku;
  while (kohdalla)
  {
    // jos valkea uhkaa mustan kuningasta valkea saa bonusta
    if(kohdalla->kohdeRivi == mustanKuninkaanPaikka[0]
    && kohdalla->kohdeSarake == mustanKuninkaanPaikka[1])
      kuningashyokkays = kuningashyokkays + 1;

    // ja sitten tutkitaan kaikki kuninkaan ympärysruudut
    if(kohdalla->kohdeRivi == mustanKuninkaanPaikka[0]+1
    && kohdalla->kohdeSarake == mustanKuninkaanPaikka[1])
      kuningashyokkays = kuningashyokkays + 1;

    if(kohdalla->kohdeRivi == mustanKuninkaanPaikka[0]+1
    && kohdalla->kohdeSarake == mustanKuninkaanPaikka[1]-1)
      kuningashyokkays = kuningashyokkays + 1;

      if(kohdalla->kohdeRivi == mustanKuninkaanPaikka[0]+1
    && kohdalla->kohdeSarake == mustanKuninkaanPaikka[1]+1)
      kuningashyokkays = kuningashyokkays + 1;

      if(kohdalla->kohdeRivi == mustanKuninkaanPaikka[0]
    && kohdalla->kohdeSarake == mustanKuninkaanPaikka[1]-1)
      kuningashyokkays = kuningashyokkays + 1;

      if(kohdalla->kohdeRivi == mustanKuninkaanPaikka[0]
    && kohdalla->kohdeSarake == mustanKuninkaanPaikka[1]+1)
      kuningashyokkays = kuningashyokkays + 1;

      if(kohdalla->kohdeRivi == mustanKuninkaanPaikka[0]-1
    && kohdalla->kohdeSarake == mustanKuninkaanPaikka[1]-1)
      kuningashyokkays = kuningashyokkays + 1;

      if(kohdalla->kohdeRivi == mustanKuninkaanPaikka[0]-1
    && kohdalla->kohdeSarake == mustanKuninkaanPaikka[1])
      kuningashyokkays = kuningashyokkays + 1;

      if(kohdalla->kohdeRivi == mustanKuninkaanPaikka[0]-1
    && kohdalla->kohdeSarake == mustanKuninkaanPaikka[1]+1)
      kuningashyokkays = kuningashyokkays + 1;
```

```
      kohdalla = kohdalla -> seuraava;
  }
  TyhjennaLista(mahdollistenSiirtojenAlku);
  mahdollistenSiirtojenAlku = NULL;
  kohdalla = NULL;

  TuotaKaikkiMahdollisetSiirrot(lauta,-
1,linnoitusoikeudet,mahdollistenSiirtojenAlku);
  kohdalla = mahdollistenSiirtojenAlku;

  while (kohdalla)
  {
    // jos musta uhkaa valkean kuningasta musta saa bonusta
    if(kohdalla->kohdeRivi == valkeanKuninkaanPaikka[0]
    && kohdalla->kohdeSarake == valkeanKuninkaanPaikka[1])
      kuningashyokkays = kuningashyokkays -1;

    // ja sitten tutkitaan kaikki kuninkaan ympärysruudut
    if(kohdalla->kohdeRivi == valkeanKuninkaanPaikka[0]+1
    && kohdalla->kohdeSarake == valkeanKuninkaanPaikka[1])
      kuningashyokkays = kuningashyokkays - 1;

    if(kohdalla->kohdeRivi == valkeanKuninkaanPaikka[0]+1
    && kohdalla->kohdeSarake == valkeanKuninkaanPaikka[1]-1)
      kuningashyokkays = kuningashyokkays - 1;

      if(kohdalla->kohdeRivi == valkeanKuninkaanPaikka[0]+1
    && kohdalla->kohdeSarake == valkeanKuninkaanPaikka[1]+1)
      kuningashyokkays = kuningashyokkays - 1;

      if(kohdalla->kohdeRivi == valkeanKuninkaanPaikka[0]
    && kohdalla->kohdeSarake == valkeanKuninkaanPaikka[1]-1)
      kuningashyokkays = kuningashyokkays - 1;

      if(kohdalla->kohdeRivi == valkeanKuninkaanPaikka[0]
    && kohdalla->kohdeSarake == valkeanKuninkaanPaikka[1]+1)
      kuningashyokkays = kuningashyokkays - 1;

      if(kohdalla->kohdeRivi == valkeanKuninkaanPaikka[0]-1
    && kohdalla->kohdeSarake == valkeanKuninkaanPaikka[1]-1)
      kuningashyokkays = kuningashyokkays - 1;

      if(kohdalla->kohdeRivi == valkeanKuninkaanPaikka[0]-1
    && kohdalla->kohdeSarake == valkeanKuninkaanPaikka[1])
      kuningashyokkays = kuningashyokkays - 1;

      if(kohdalla->kohdeRivi == valkeanKuninkaanPaikka[0]-1
    && kohdalla->kohdeSarake == valkeanKuninkaanPaikka[1]+1)
      kuningashyokkays = kuningashyokkays - 1;

      kohdalla = kohdalla -> seuraava;
  }

  TyhjennaLista(mahdollistenSiirtojenAlku);
  mahdollistenSiirtojenAlku = NULL;
  return kuningashyokkays;
}


double KuninkaanTurvallisuus(int lauta[12][12])
{
  double kuninkaanTurvallisuus=0;
  int rivi, sarake;
```

```
  for (rivi = 2; rivi < 10; rivi ++)
    for (sarake = 2; sarake < 10; sarake++)
    {
    // lasketaan mustan kuninkaan turvallisuus
    // musta saa bonusta kustakin sotilaasta, joka on suoraan kuninkaan
    // edessä, edessä oikealla tai edessä vasemmalla
    if(lauta[rivi][sarake]==-6) {
      if (lauta[rivi+1][sarake-1]==-1)
        kuninkaanTurvallisuus = kuninkaanTurvallisuus - 1;
      if (lauta[rivi+1][sarake]==-1)
        kuninkaanTurvallisuus = kuninkaanTurvallisuus - 1;
      if (lauta[rivi+1][sarake+1]==-1)
        kuninkaanTurvallisuus = kuninkaanTurvallisuus - 1;
    }
    // lasketaan valkean kuninkaan turvallisuus
    // musta saa bonusta kustakin sotilaasta, joka on suoraan kuninkaan
    // edessä, edessä oikealla tai edessä vasemmalla
      if(lauta[rivi][sarake]==6) {
      if (lauta[rivi-1][sarake-1]==1)
        kuninkaanTurvallisuus = kuninkaanTurvallisuus + 1;
      if (lauta[rivi-1][sarake]==1)
        kuninkaanTurvallisuus = kuninkaanTurvallisuus + 1;
      if (lauta[rivi-1][sarake+1]==1)
        kuninkaanTurvallisuus = kuninkaanTurvallisuus + 1;
    }

}
return kuninkaanTurvallisuus;
}

double Sotilasasema(int lauta[12][12])
{
  double sotilasasema=0;
  int rivi, sarake;
  int apurivi;
  for (rivi = 2; rivi < 10; rivi ++)
    for (sarake = 2; sarake < 10; sarake++)
    {
    // etsitään valkean sotilaat
    if (lauta[rivi][sarake] == 1) {
      for(apurivi=2; apurivi < 10; apurivi++)
        // tutkitaan onko muita omia sotilaita samalla sarakkeella
        if(lauta[apurivi][sarake]==1 && apurivi != rivi)
          sotilasasema = sotilasasema - 3;
          // -3 yksikköä on sekä Kotokin että Levyn antama heikennys pelkästä
          // kaksoissotilaasta
      }
     // ja vastaava haku mustille sotilailla
     if (lauta[rivi][sarake] == -1) {
      for(apurivi=2; apurivi < 10; apurivi++)
        // tutkitaan onko muita omia sotilaita samalla sarakkeella
        if(lauta[apurivi][sarake]==-1 && apurivi != rivi)
          sotilasasema = sotilasasema + 3;
          // -3 yksikköä on sekä Kojakin että Levyn antama heikennys pelkästä
          // kaksoissotilaasta
     }
}
return sotilasasema;
}

double Kehitys(int lauta[12][12])
{
double mustanNappulatPaikoiltaanPois = 0;
double valkeanNappulatPaikoiltaanPois = 0;
```

```
double valkeanNappulatPaikoillaan = 0;
double mustanNappulatPaikoillaan = 0;
double kehitys = 0.0;
  // lasketaan kuinka moni mustan kevyt upseeri on poissa alkuperäiseltä
paikaltaan
  if (lauta[2][3] != -2)
    mustanNappulatPaikoiltaanPois = mustanNappulatPaikoiltaanPois + 1;
  if (lauta[2][4] != -3)
    mustanNappulatPaikoiltaanPois = mustanNappulatPaikoiltaanPois + 1;
  if (lauta[2][7] != -3)
    mustanNappulatPaikoiltaanPois = mustanNappulatPaikoiltaanPois + 1;
  if (lauta[2][8] != -2)
    mustanNappulatPaikoiltaanPois = mustanNappulatPaikoiltaanPois + 1;

  // lasketaan kuinka moni mustan kevyt upseeri pois paikoiltaan
  if (lauta[9][3] != 2)
    valkeanNappulatPaikoiltaanPois = valkeanNappulatPaikoiltaanPois + 1;
  if (lauta[9][4] != 3)
    valkeanNappulatPaikoiltaanPois = valkeanNappulatPaikoiltaanPois + 1;
  if (lauta[9][7] != 3)
    valkeanNappulatPaikoiltaanPois = valkeanNappulatPaikoiltaanPois + 1;
  if (lauta[9][8] != 2)
    valkeanNappulatPaikoiltaanPois = valkeanNappulatPaikoiltaanPois + 1;

  // lasketaan kehitys joka suosii siirtämään ensin kevyitä upseereita
  kehitys = (valkeanNappulatPaikoiltaanPois - mustanNappulatPaikoiltaanPois)/3;
  // seuraavaksi sakotetaan liian aikaisesta daamin siirtämistä
  // käytännön kokeilut melkein pakottivat tämän kehityselementin
  // lisäämisen sillä daami saa niin paljon liikkumispisteitä että kone
  // hyvin mielellään siirtää sitä heti kun mahdollista

  // lasketaan mustan nappulat(raskaat ja kevyet) jotka ovat paikoillaan
  if(lauta[2][5]==-5)
    mustanNappulatPaikoillaan=0;
  else
  {
  if (lauta[2][2]==-4)
    mustanNappulatPaikoillaan = mustanNappulatPaikoillaan+1;
  if (lauta[2][3]==-2)
    mustanNappulatPaikoillaan = mustanNappulatPaikoillaan+1;
  if (lauta[2][4]==-3)
    mustanNappulatPaikoillaan = mustanNappulatPaikoillaan+1;
  if (lauta[2][7]==-3)
    mustanNappulatPaikoillaan = mustanNappulatPaikoillaan+1;
  if (lauta[2][8]==-2)
    mustanNappulatPaikoillaan = mustanNappulatPaikoillaan+1;
  if (lauta[2][9]==-4)
    mustanNappulatPaikoillaan = mustanNappulatPaikoillaan+1;
  }
  // ja vastaavasti valkean
  if(lauta[9][5]==5)
    valkeanNappulatPaikoillaan = 0;
  else
  {
  if (lauta[9][2]==4)
    valkeanNappulatPaikoillaan = valkeanNappulatPaikoillaan + 1;
  if (lauta[9][3]==2)
    valkeanNappulatPaikoillaan = valkeanNappulatPaikoillaan + 1;
  if (lauta[9][4]==3)
    valkeanNappulatPaikoillaan = valkeanNappulatPaikoillaan + 1;
  if (lauta[9][7]==3)
    valkeanNappulatPaikoillaan = valkeanNappulatPaikoillaan + 1;
  if (lauta[9][7]==2)
    valkeanNappulatPaikoillaan = valkeanNappulatPaikoillaan + 1;
```

```
  if (lauta[9][9]==4)
    valkeanNappulatPaikoillaan = valkeanNappulatPaikoillaan + 1;
  }
      kehitys = kehitys - (valkeanNappulatPaikoillaan/4 -
mustanNappulatPaikoillaan/4);

return kehitys;
}

double KeskustanHallinta(int lauta[12][12])
{
double keskustanHallinta=0;

    // keskustaruudun d5 hallinnastaa saa yhden yksikön
    if(lauta[5][5] < 0)
      keskustanHallinta = keskustanHallinta - 1;
    if(lauta[5][5] > 0)
      keskustanHallinta = keskustanHallinta + 1;

    // keskustaruudun e5 hallinnasta saa yhden yksikön
    if(lauta[5][6] < 0)
      keskustanHallinta = keskustanHallinta - 1;
    if(lauta[5][6] > 0)
      keskustanHallinta = keskustanHallinta + 1;

    // keskustaruuden d4 hallinnasta saa yhden yksikön
    if(lauta[6][5] < 0)
      keskustanHallinta = keskustanHallinta - 1;
    if(lauta[6][5] > 0)
      keskustanHallinta = keskustanHallinta + 1;

    // keskustaruudun e4 hallinnasta saa yhden yksikön
    if(lauta[6][6] < 0)
      keskustanHallinta = keskustanHallinta - 1;
    if(lauta[6][6] > 0)
      keskustanHallinta = keskustanHallinta + 1;

return keskustanHallinta;
}

double LaskeMateriaali(int lauta[12][12])
{
double asemanArvo = 0.00;
  int rivi, sarake;
  for (rivi = 2; rivi < 10; rivi ++)

    for (sarake = 2; sarake < 10; sarake++)
    {
      if (lauta[rivi][sarake]==-1)
        asemanArvo=asemanArvo-1;
      if (lauta[rivi][sarake]==-2)
        asemanArvo=asemanArvo-3;
      if (lauta[rivi][sarake]==-3)
        asemanArvo=asemanArvo-3.25;
      if (lauta[rivi][sarake]==-4)
        asemanArvo=asemanArvo-5;
      if (lauta[rivi][sarake]==-5)
        asemanArvo=asemanArvo-9;
      if (lauta[rivi][sarake]==-6)
        asemanArvo=asemanArvo-1000;
      if (lauta[rivi][sarake]==1)
        asemanArvo=asemanArvo+1;
        if (lauta[rivi][sarake]==2)
        asemanArvo=asemanArvo+3;
```

```
      if (lauta[rivi][sarake]==3)
        asemanArvo=asemanArvo+3.25;
      if (lauta[rivi][sarake]==4)
        asemanArvo=asemanArvo+5;
      if (lauta[rivi][sarake]==5)
        asemanArvo=asemanArvo+9;
      if (lauta[rivi][sarake]==6)
        asemanArvo=asemanArvo+1000;
  }
return asemanArvo;
}

double ArvioiAsema(int lauta[12][12],int linnoitusoikeudet[4], int siirtovuoro)
{
  double asemanArvo = 0.00;
  double liikkuvuus = 0;
  double valkeanLiikkuvuus=0;
  double mustanLiikkuvuus=0;

  int tilanne;

  Siirto *laillistenSiirtojenAlku = NULL;
  // tuotetaan valkean kaikki lailliset siirrot

TuotaKaikkiLaillisetSiirrot(lauta,1,linnoitusoikeudet,laillistenSiirtojenAlku);
  Siirto *kohdalla;
  kohdalla = laillistenSiirtojenAlku;
  while (kohdalla)
  {
    valkeanLiikkuvuus++;
    kohdalla = kohdalla->seuraava;
  }
  TyhjennaLista(laillistenSiirtojenAlku);
  // tuotetaan mustan kaikki lailliset siirrot
  TuotaKaikkiLaillisetSiirrot(lauta,-
1,linnoitusoikeudet,laillistenSiirtojenAlku);
  kohdalla = laillistenSiirtojenAlku;

    while (kohdalla)
  {
    mustanLiikkuvuus++;
    kohdalla = kohdalla->seuraava;
  }
  // lasketaan liikkuvuusero
  liikkuvuus = valkeanLiikkuvuus - mustanLiikkuvuus;
  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku = NULL;
  kohdalla = NULL;

  if (valkeanLiikkuvuus == 0 || mustanLiikkuvuus == 0)
  {
    tilanne = TarkistaOnkoMatti(lauta);
    if (tilanne==1000)
      return 1000;
    if (tilanne==-1000)
      return -1000;
    if (tilanne==0 && valkeanLiikkuvuus==0 && siirtovuoro==1)
      return 0;
    if (tilanne==0 && mustanLiikkuvuus==0 && siirtovuoro==-1)
      return 0;
   }

  asemanArvo =     LaskeMateriaali(lauta)
             +0.03*liikkuvuus
```

```
            +0.10*KeskustanHallinta(lauta)
            +0.55*Kehitys(lauta)
            +0.035*Sotilasasema(lauta)
            +0.10*KuninkaanTurvallisuus(lauta)
            +0.07*KuningasHyokkays(lauta);
return asemanArvo;
}

double MiniMaxAlfaBeta(int syvyys,double alfa,double beta, int parasSiirto[5],
int asetettuSyvyys, int lauta[12][12],int siirtovuoro,int siirtovuoroLopussa,
int linnoitusoikeudet[4])
{
  if (syvyys == 0) {
    asemaLaskuri++;
    return siirtovuoroLopussa*ArvioiAsema(lauta,linnoitusoikeudet,siirtovuoro);
  }

  double asemanArvo = 0.0;
  Siirto *laillistenSiirtojenAlku = NULL;
  TyhjennaLista(laillistenSiirtojenAlku);
  int muistiAlkuNappula;
  int muistiKohdeNappula;


TuotaKaikkiLaillisetSiirrot(lauta,siirtovuoro,linnoitusoikeudet,laillistenSiirto
jenAlku); //

   // jos siirtovuorossa olevalla pelaajalla ei ole laillisia siirtoja
   // pelaaja on joko matissa tai patissa ja nyt tarkistetaan kummasta
   // on kyse ja samalla luonnollisesti hakupuuta ei tarvitse tästä
   // asemasta enää laskea syvämmälle vaan voidaan katkaista haku ja
   // arvona säilyy 1000, jos valkea on tehnyt matin, -1000, jos musta on
   // tehnyt matin ja 0, jos tilanne on patti
   if (laillistenSiirtojenAlku == NULL)
     return TarkistaOnkoMatti(lauta);

  Siirto *kohdalla;
  kohdalla = laillistenSiirtojenAlku;
  while (kohdalla)
  {
    muistiAlkuNappula = lauta[kohdalla->alkuRivi][kohdalla->alkuSarake];
    muistiKohdeNappula = lauta[kohdalla->kohdeRivi][kohdalla->kohdeSarake];

    TeeSiirto(kohdalla->alkuRivi, kohdalla->alkuSarake,kohdalla-
>kohdeRivi,kohdalla->kohdeSarake,kohdalla->korotusNappula,lauta); // ALK PER
    asemanArvo = -MiniMaxAlfaBeta(syvyys-1, -beta, -alfa, parasSiirto,
asetettuSyvyys,lauta,siirtovuoro*-1,siirtovuoroLopussa,linnoitusoikeudet);
    KumoaSiirto(kohdalla->alkuRivi, kohdalla->alkuSarake, kohdalla->kohdeRivi,
kohdalla->kohdeSarake, muistiAlkuNappula, muistiKohdeNappula,lauta);

    if (asemanArvo >= beta) {
    TyhjennaLista(laillistenSiirtojenAlku);
    laillistenSiirtojenAlku = NULL;
    kohdalla = NULL;
      return beta; }

    if (asemanArvo > alfa) {
      alfa = asemanArvo;
      if (asetettuSyvyys == syvyys) {
      parasSiirto[0] = kohdalla->alkuRivi;
      parasSiirto[1] = kohdalla->alkuSarake;
      parasSiirto[2] = kohdalla->kohdeRivi;
      parasSiirto[3] = kohdalla->kohdeSarake;
      parasSiirto[4] = kohdalla->korotusNappula;
```

```
      }
    }
    kohdalla = kohdalla->seuraava;
  }
  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku = NULL;
  kohdalla = NULL;
return alfa;
}

int PaaValikko()
{
  int vastaus;
  cout << "\n\nPÄÄVALIKKO";
  cout << "\n\n1) Uusi peli";
  cout << "\n2) Ohjeet";
  cout << "\n0) Lopetus\n";
  cin >> vastaus;
  return vastaus;
}

int main(void)
{

   int avaussiirrot[3][5] = {{8,6,6,6,0},
                             {8,5,6,5,0},
                             {8,4,6,4,0}};

   int vastaukset_e4[3][5] = {{3,6,5,6,0},
                             {3,4,5,4,0},
                             {3,6,4,6,0}};


  double asemanArvo = 0.00;
  int siirtovuoro = 1;
  int siirtovuoroLopussa=1;
  int pariton = 1;
  int siirtoLaskuri = 1;

  int linnoitusoikeudet[4] = {1,1,1,1}; // linnoitusoikeudet voimaan

  int parasSiirto[5] = {0,0,0,0,0};
  int syvyys = 0;
  int asetettuSyvyys = 0;

  Siirto *laillistenSiirtojenAlku = NULL;

  int pelaajanAlkuRivi=1;
  int pelaajanAlkuSarake=0;
  int pelaajanKohdeRivi=0;
  int pelaajanKohdeSarake=0;
  int pelaajanKorotusNappula=0;
  char pelaajanKorotusNappulaMerkki = ' ';

  char pAlkuRivi= ' ';
  char pAlkuSarake= ' ';
  char pKohdeRivi= ' ';
  char pKohdeSarake= ' ';


  int tarkistus = 0;
  int apu=0;

  int satunnaisLuku;
```

```
asemaLaskuri = 0;
int vaikeustaso = 0;
int vastaus = 99;
int vari = 0;

printf("\n\nPERUSSHAKKI\n");
cout << "\nAki Seppälä, 2004";

  vastaus = PaaValikko();
while (vastaus != 0) {

  if (vastaus==2) {
  cout << "\nOHJEET";
  cout << "\n Anna lähtöruudut muodossa [linja][rivi] esim. e2 tai g1";
  cout << "\n Anna kohderuudut muodossa [linja][rivi] esim. e4 tai f3";
  cout << "\n Jos sotilaasi korottuu, valitse korotusnappulaksi D,T,L tai R";
  cout << "\n Anna lähtöruuduksi 0, jos haluat lopettaa pelin";
  }

  if (vastaus==1) {

  int lauta[12][12] = {{99,99,99,99,99,99,99,99,99,99,99,99},
                       {99,99,99,99,99,99,99,99,99,99,99,99},
                       {99,99,-4,-2,-3,-5,-6,-3,-2,-4,99,99},
                       {99,99,-1,-1,-1,-1,-1,-1,-1,-1,99,99},
                       {99,99,0,0,0,0,0,0,0,0,99,99},
                       {99,99,0,0,0,0,0,0,0,0,99,99},
                       {99,99,0,0,0,0,0,0,0,0,99,99},
                       {99,99,0,0,0,0,0,0,0,0,99,99},
                       {99,99,1,1,1,1,1,1,1,1,99,99},
                       {99,99,4,2,3,5,6,3,2,4,99,99},
                       {99,99,99,99,99,99,99,99,99,99,99,99},
                       {99,99,99,99,99,99,99,99,99,99,99,99}};

    while (vaikeustaso != 1
    && vaikeustaso != 2
    && vaikeustaso != 3
    && vaikeustaso != 4
    && vaikeustaso != 5
    && vaikeustaso != 6
    && vaikeustaso != 7
    && vaikeustaso != 8)
    {
      cout << "\nVALITSE VAIKEUSTASO";
      cout << "\n1) Todella helppo";
      cout << "\n2) Helppo";
      cout << "\n3) Normaali";
      cout << "\n4) Melko vaikea";
      cout << "\n5) Vaikea";
      cout << "\n6) Todella vaikea";
      cout << "\n7) Supervaikea ";
      cout << "\n8) Hypersupervaikea\n";
      cin >> vaikeustaso;
    }
    while (vari != 1 && vari != 2)
    {
      cout << "\nValitse kummilla väreillä haluat pelata";
      cout << "\n1) Valkeilla";
      cout << "\n2) Mustilla\n";
      cin >> vari;
    }

    if (vaikeustaso == 1 && vari == 1)
```

```
{
   asetettuSyvyys = 1;         // vaikeustaso 1 => 1 puolisiirron haku

   siirtovuoro = 1;            // pelaaja valkeilla

   siirtovuoroLopussa = 1;     // siirtovuoro hakupuun lehtisolmuissa

   pariton = -1;               // tällä tahdistetaan aseman arvot niin
                               // että plus merkkinen arvo tarkoittaa
                               // aina valkean etua ja miinusmerkkinen
                               // mustan etua
}
if (vaikeustaso == 1 && vari == 2)
{
  asetettuSyvyys = 1;          // vaikeustaso 1 => 1 puolisiirron haku
  siirtovuoro = -1;            // pelaaja mustilla
  siirtovuoroLopussa = -1;
  pariton = -1;
}
if (vaikeustaso == 2 && vari == 1)
{
  asetettuSyvyys = 2;            // hakusyvyys 2
  siirtovuoro = 1;               // pelaaja valkeilla
  siirtovuoroLopussa = -1;
  pariton = 1;
}

if (vaikeustaso == 2 && vari == 2)
{
asetettuSyvyys = 2;            // hakusyvyys 2
siirtovuoro = -1;              // pelaaja mustilla
siirtovuoroLopussa = 1;
pariton = 1;
}
if (vaikeustaso == 3 && vari == 1)
{
asetettuSyvyys = 3;         // hakusyvyys 3
siirtovuoro = 1;            // pelaaja valkeilla
siirtovuoroLopussa = 1;
pariton = -1;
}
if (vaikeustaso == 3 && vari == 2)
{
  asetettuSyvyys = 3;          // hakusyvyys 3
  siirtovuoro = -1;            // pelaaja mustilla
  siirtovuoroLopussa = -1;
  pariton = -1;
}

if (vaikeustaso == 4 && vari == 1)
{
  asetettuSyvyys = 4;            // hakusyvyys 4
  siirtovuoro = 1;               // pelaaja valkeilla
  siirtovuoroLopussa = -1;
  pariton = 1;
}

if (vaikeustaso == 4 && vari == 2)
{
asetettuSyvyys = 4;            // hakusyvyys 4
siirtovuoro = -1;              // pelaaja mustilla
siirtovuoroLopussa = 1;
pariton = 1;
}
```

```
    if (vaikeustaso == 5 && vari == 1)
    {
    asetettuSyvyys = 5;         // hakusyvyys 5
    siirtovuoro = 1;            // pelaaja valkeilla
    siirtovuoroLopussa = 1;
    pariton = -1;
    }
    if (vaikeustaso == 5 && vari == 2)
    {
      asetettuSyvyys = 5;          // hakusyvyys 5
      siirtovuoro = -1;            // pelaaja mustilla
      siirtovuoroLopussa = -1;
      pariton = -1;
    }

    if (vaikeustaso == 6 && vari == 1)
    {
    asetettuSyvyys = 7;         // hakusyvyys 7
    siirtovuoro = 1;            // pelaaja valkeilla
    siirtovuoroLopussa = 1;
    pariton = -1;
    }
    if (vaikeustaso == 6 && vari == 2)
    {
      asetettuSyvyys = 7;          // hakusyvyys 7
      siirtovuoro = -1;            // pelaaja mustilla
      siirtovuoroLopussa = -1;
      pariton = -1;
    }

    if (vaikeustaso == 7 && vari == 1)
    {
    asetettuSyvyys = 9;         // hakusyvyys 9
    siirtovuoro = 1;            // pelaaja valkeilla
    siirtovuoroLopussa = 1;
    pariton = -1;
    }
    if (vaikeustaso == 7 && vari == 2)
    {
      asetettuSyvyys = 9;          // hakusyvyys 9
      siirtovuoro = -1;            // pelaaja mustilla
      siirtovuoroLopussa = -1;
      pariton = -1;
    }

    if (vaikeustaso == 8 && vari == 1)
    {
    asetettuSyvyys = 11;         // hakusyvyys 11
    siirtovuoro = 1;            // pelaaja valkeilla
    siirtovuoroLopussa = 1;
    pariton = -1;
    }
    if (vaikeustaso == 8 && vari == 2)
    {
      asetettuSyvyys = 11;          // hakusyvyys 11
      siirtovuoro = -1;            // pelaaja mustilla
      siirtovuoroLopussa = -1;
      pariton = -1;
    }

  randomize();
  if (siirtovuoro==-1) {
    randomize();
```

```
    satunnaisLuku = random(3);  // satunnaisluku väliltä 0-2
    TeeSiirto(avaussiirrot[satunnaisLuku][0]
     ,avaussiirrot[satunnaisLuku][1]
     ,avaussiirrot[satunnaisLuku][2]
     ,avaussiirrot[satunnaisLuku][3]
     ,avaussiirrot[satunnaisLuku][4], lauta);
  }
TulostaLauta(lauta);

    while (tarkistus == 0 && pAlkuRivi != '0')

  {

 int virhe = 0;

    do
  {
    laillistenSiirtojenAlku = NULL;
    TyhjennaLista(laillistenSiirtojenAlku);
    laillistenSiirtojenAlku = NULL;

TuotaKaikkiLaillisetSiirrot(lauta,siirtovuoro,linnoitusoikeudet,laillistenSiirto
jenAlku);

    if (virhe == 1)
      cout << "\nAnnoit laittoman siirron. Yritä uudestaan.";
    cout<<"\nAnna siirtosi koordinaatit.\n";
    cout<<"\Lähtöruutu:";
    cin>>pAlkuRivi;
    if (pAlkuRivi == '0')
      break;
    cin>>pAlkuSarake;
    cout<<"\Kohderuutu:";
    cin>>pKohdeRivi;
    cin>>pKohdeSarake;
    virhe = 1;

    pelaajanAlkuRivi = Muunnos(pAlkuSarake);
    pelaajanAlkuSarake = Muunnos(pAlkuRivi);
    pelaajanKohdeRivi = Muunnos(pKohdeSarake);
    pelaajanKohdeSarake = Muunnos(pKohdeRivi);

  }
  while
(TarkistaPelaajanSiirto(pelaajanAlkuRivi,pelaajanAlkuSarake,pelaajanKohdeRivi,pe
laajanKohdeSarake,laillistenSiirtojenAlku)==0 && pAlkuRivi != '0');
  TyhjennaLista(laillistenSiirtojenAlku);
  if (pAlkuRivi != '0')
  {
  // jos sotilas siirtyy nollataan laskuri joka kyttää
  // ollaanko tehty 50 siirtoa ilman sotilaiden siirtoa
  // joilloin asema olisi tasapeli

  if (lauta[pelaajanAlkuRivi][pelaajanAlkuSarake]==1
  || lauta[pelaajanAlkuRivi][pelaajanAlkuSarake]==-1)
    siirrotIlmanSotilaidenSiirtoja = 0;

  pelaajanKorotusNappula=0;

  // tutkitaan onko kyseessä valkean sotilaan korotus
  if (lauta[pelaajanAlkuRivi][pelaajanAlkuSarake]==1
  && pelaajanKohdeRivi==2)
  {
    cout << "Anna korotusnappula:";
```

```
    cin>>pelaajanKorotusNappulaMerkki;
    if (pelaajanKorotusNappulaMerkki =='D')
      pelaajanKorotusNappula = 5;
    if (pelaajanKorotusNappulaMerkki =='T')
      pelaajanKorotusNappula = 4;
    if (pelaajanKorotusNappulaMerkki =='L')
      pelaajanKorotusNappula = 3;
    if (pelaajanKorotusNappulaMerkki =='R')
      pelaajanKorotusNappula = 2;
  }
  // tutkitaan onko musta sotilas korottumassa
  if (lauta[pelaajanAlkuRivi][pelaajanAlkuSarake]==-1
  && pelaajanKohdeRivi==9)
  {
    cout << "Anna korotusnappula:";
    cin>>pelaajanKorotusNappulaMerkki;
    if (pelaajanKorotusNappulaMerkki =='D')
      pelaajanKorotusNappula = -5;
    if (pelaajanKorotusNappulaMerkki =='T')
      pelaajanKorotusNappula = -4;
    if (pelaajanKorotusNappulaMerkki =='L')
      pelaajanKorotusNappula = -3;
    if (pelaajanKorotusNappulaMerkki =='R')
      pelaajanKorotusNappula = -2;
  }

TeeSiirto(pelaajanAlkuRivi,pelaajanAlkuSarake,pelaajanKohdeRivi,pelaajanKohdeSar
ake,pelaajanKorotusNappula,lauta);

  // kielletään tarvittaessa linnoitusoikeuksia

  // jos valkea kuningas liikkunut valkea menettää kaikki linnoitusoikeudet
  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake]==6)
  {
    linnoitusoikeudet[0]=0;
    linnoitusoikeudet[1]=0;
  }
  // jos musta kuningas liikkunut musta menettää kaikki linnoitusoikeudet
  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake]==-6)
  {
    linnoitusoikeudet[2]=0;
    linnoitusoikeudet[3]=0;
  }
  // jos valkean torni liikkuu valkea menettää sen puolisen linnoituisoikeuden
  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake]==4 && pelaajanAlkuSarake==2)
    linnoitusoikeudet[1]=0;
  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake]==4 && pelaajanAlkuSarake==9)
    linnoitusoikeudet[0]=0;

  // samoin musta
  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake]==-4 &&
pelaajanAlkuSarake==2)
    linnoitusoikeudet[3]=0;
  if (lauta[pelaajanKohdeRivi][pelaajanKohdeSarake]==-4 &&
pelaajanAlkuSarake==9)
    linnoitusoikeudet[2]=0;


  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku = NULL;
  TulostaLauta(lauta);
  tarkistus = TarkistaTilanne(lauta,siirtovuoro);
  if (tarkistus == 0)
  {
```

```
  siirtovuoro = siirtovuoro*-1;

  syvyys = asetettuSyvyys;

  // tutkitaan ensin löytyykö vastaus avauskirjastosta

  if (siirtoLaskuri==1
    && pelaajanAlkuRivi == avaussiirrot[0][0]
    && pelaajanAlkuSarake == avaussiirrot[0][1]
    && pelaajanKohdeRivi == avaussiirrot[0][2]
    && pelaajanKohdeSarake == avaussiirrot[0][3])
    {

      satunnaisLuku = random(3);
      TeeSiirto(vastaukset_e4[satunnaisLuku][0]
     ,vastaukset_e4[satunnaisLuku][1]
     ,vastaukset_e4[satunnaisLuku][2]
     ,vastaukset_e4[satunnaisLuku][3]
     ,vastaukset_e4[satunnaisLuku][4], lauta);
     cout << "\n";
    }
  else
  {
  // tutkitaan ensin saako musta yhden siirron matin
  // sen lisäksi että tämä säästää aikaa mattitapauksissa se estää
  // myös oravanpyörän missä ohjelma näkee matin 1 siirron syvydessä
  // ja esim. 3 siirron syvyydessä ja lähtee aina 3 siirron syvyyden
mattimuunnelmaan
  // eikä saa oikeasti mattia ikinä tehtyä
  apu = NappulaMuunnos(lauta, pelaajanKohdeRivi, pelaajanKohdeSarake);
  cout << "\n Siirsit " << siirtoLaskuri;
  cout << ".";
  if (vari==2)
    cout <<" - ";
  cout << NappulaMuunnos(lauta, pelaajanKohdeRivi, pelaajanKohdeSarake);
  cout << MuunnosSarakeRivi(pelaajanKohdeSarake);
  cout << MuunnosRiviSarake(pelaajanKohdeRivi);

  cout << "\n\nOhjelma miettii siirtoaan...";
  asemanArvo = siirtovuoro*MiniMaxAlfaBeta(1,-2000, 2000, parasSiirto,
1,lauta,siirtovuoro,siirtovuoroLopussa,linnoitusoikeudet);
  if ((asemanArvo > -500 && asemanArvo < 500) && asetettuSyvyys > 1)
  {
     asemanArvo = siirtovuoro*MiniMaxAlfaBeta(syvyys,-2000, 2000, parasSiirto,
asetettuSyvyys,lauta,siirtovuoro,siirtovuoroLopussa,linnoitusoikeudet);
  }
  cout << "\nOhjelmalle paras mahdollinen aseman arvo ";
  cout << syvyys ;
  cout << " puolisiirron jälkeen:";
  printf("%6.2lf", pariton*siirtovuoro*siirtovuoroLopussa*asemanArvo);
  cout <<"\n";
  cout << "Paras mahdollinen siirto: ";
  if (vari==1) {
  cout << siirtoLaskuri;
  cout << ". - ";
  }
  else {
    apu = siirtoLaskuri+1;
  cout << apu;
  cout << ".";
  }
  cout << NappulaMuunnos(lauta,parasSiirto[0],parasSiirto[1]);
  cout << MuunnosSarakeRivi(parasSiirto[3]);
  cout << MuunnosRiviSarake(parasSiirto[2]);
```

```
  cout << "\nAsemia arvioitu: ";
  cout << asemaLaskuri;

  if (lauta[parasSiirto[0]][parasSiirto[1]]==1
  || lauta[parasSiirto[0]][parasSiirto[1]]==-1)
    siirrotIlmanSotilaidenSiirtoja = 0;


TeeSiirto(parasSiirto[0],parasSiirto[1],parasSiirto[2],parasSiirto[3],parasSiirt
o[4],lauta);

  if (lauta[parasSiirto[2]][parasSiirto[3]]==6)
  {
    linnoitusoikeudet[0]=0;
    linnoitusoikeudet[1]=0;
  }
  if (lauta[parasSiirto[2]][parasSiirto[3]]==-6)
  {
    linnoitusoikeudet[2]=0;
    linnoitusoikeudet[3]=0;
  }
  if (lauta[parasSiirto[2]][parasSiirto[3]]==4 && parasSiirto[1]==2)
    linnoitusoikeudet[1]=0;
  if (lauta[parasSiirto[2]][parasSiirto[3]]==4 && parasSiirto[1]==9)
    linnoitusoikeudet[0]=0;

  if (lauta[parasSiirto[2]][parasSiirto[3]]==-4 && parasSiirto[1]==2)
    linnoitusoikeudet[3]=0;
  if (lauta[parasSiirto[2]][parasSiirto[3]]==-4 && parasSiirto[1]==9)
    linnoitusoikeudet[2]=0;

  }
 TulostaLauta(lauta);
  asemanArvo=ArvioiAsema(lauta,linnoitusoikeudet,siirtovuoro);
  cout <<"\nNykyisen aseman arvo: ";
  printf("%6.2lf",asemanArvo);
  siirtovuoro = siirtovuoro*-1;
  asemaLaskuri = 0;
  parasSiirto[0] = 0;
  parasSiirto[1] = 0;
  parasSiirto[2] = 0;
  parasSiirto[3] = 0;
  parasSiirto[4] = 0;
  TyhjennaLista(laillistenSiirtojenAlku);
  laillistenSiirtojenAlku = NULL;
  tarkistus = TarkistaTilanne(lauta,siirtovuoro);
  siirtoLaskuri++;
  siirrotIlmanSotilaidenSiirtoja++;
  }
  }
  }

  if (tarkistus==-1)
    cout << "\nMusta teki matin! (0-1)";
  if (tarkistus== 1)
    cout << "\nValkea teki matin! (1-0)";
  if (tarkistus == 2)
    cout << "\nPatti! Tasapeli (½-½).";
  if (tarkistus == 3)
    cout << "\nMateriaali ei riitä matin tekemiseen. Tasapeli (½-½).";
  if (tarkistus == 4)
    cout << "\n50 siirtoa ilman sotilaiden siirtoja. Tasapeli (½-½).";
  }
```

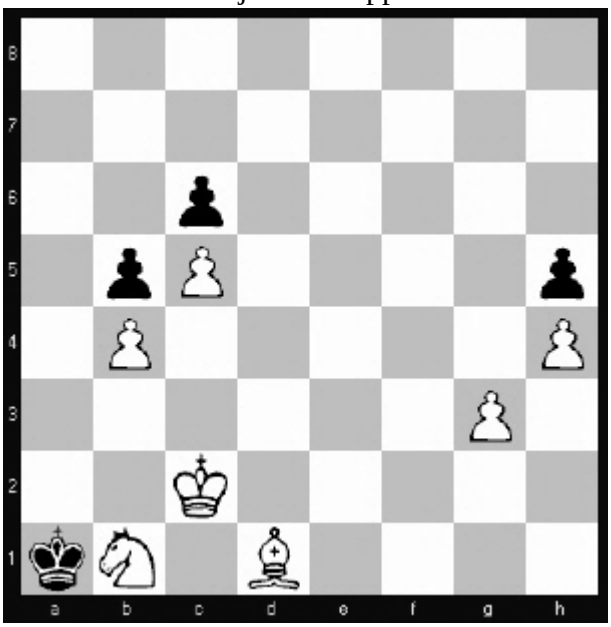
```
  tarkistus = 0;
  vastaus = 99;
  vaikeustaso = 0;
  vari = 0;
  pAlkuRivi = ' ';
  siirtoLaskuri = 1;
  siirrotIlmanSotilaidenSiirtoja=0;
  linnoitusoikeudet[1] = 1;
  linnoitusoikeudet[2] = 1;
  linnoitusoikeudet[3] = 1;
  linnoitusoikeudet[4] = 1;

  vastaus = PaaValikko();
  }
  cout << "\nKiitos pelistä. Tervetuloa uudelleen pelaamaan Perus-Shakkia!";

  return 0;
}
```

Liite 2. Mattitehtävien ratkaisut.

Laatija: Aki Seppälä



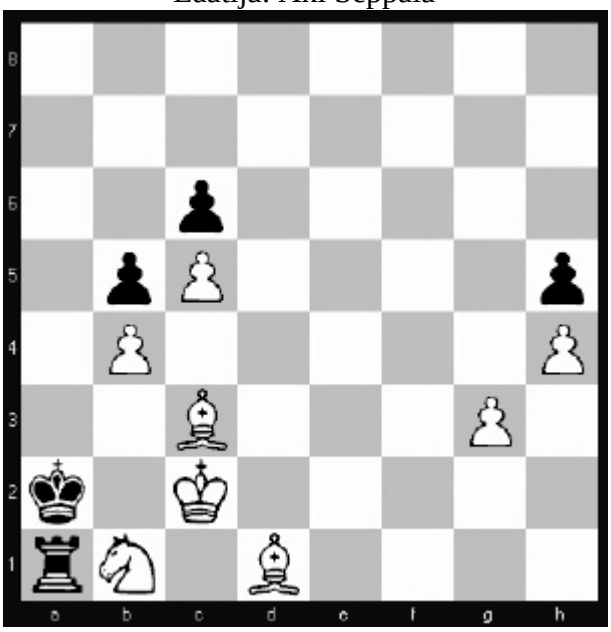
6X 6+4

Selostus: Tässä ensimmäisessä versiossa kaikkien versioiden perusideana oleva lähetin uhraus (ja tornin korotus) tulee kauneimmin esiin. Tehtävässä ei kuitenkaan ole paljoa houkutuksia ja lisäksi tehtävä ei ole täysin korrekti, koska tehtävä sisältää dualismin eli siirtojen 4.Ra3 ja 5.Kc1 järjestystä voi vaihtaa.

Ratkaisu: Houkutus: 1.g4? hxg4 2.h5 g3 3.h6 g2 4.h7 g1D 5.h8D+ Dg7! 6.Da8+ Da7! 7.Dxa7X

Oikein on: **1.Lg4!!** (1.Lxh5? ei toimi, koska silloin musta pelastautuu pattiin h5-sotilaan puuttumisen ansiosta) **1. - Ka2 (1. - hxg4 2. h5 Ka2 3.h6 Ka1 4.h7 Ka2 5.h8T!**(5.h8D?) Patti) **Ka1 6. Ta8X)2.Le6+ Kh1 3.g4!** (estää patin) **hxg4 4.Ra3 g3 5.Kc1 g2 6.Rc2X**

Laatija: Aki Seppälä



7X 8 + 5

Selostus: Tässä välivaiheen versiossa on edellisen version ideoiden lisäksi kaksi aivan uudenlaista ideaa, joita ei liene ole ennen nähty mattitehtävissä. Yleinen shakkisääntö sanoo, että lähetillä ja ratsulla voi tehdä matin vain lähetin väriseen nurkkaan. Tämä tehtävä sisältää opetuksen, että erikoistapauksissa matin voi tehdä myös ”väärän” väriseen nurkkaan. Tässähän täytyy heti aluksi tehdä strateginen valinta lähteekö tekemään mattia lähettiparilla (alkaen 1.Lxh5), ratsulla ja “oikean” värisellä lähetillä (1.Rd2) vai ratsulla ja “väärän” värisellä lähetillä (1.Lxa1). Mattitehtävissä ei yleensä syödä ensimmäisellä siirrolla ja ensimmäinen siirto on kaunein. Tässä tehtävässä sen sijaan syödään ensimmäisellä siirrolla ja toinen siirto on kaunein. Nämä kaikki ideat tekevät tehtävästä vaikean ratkaistavan. Suomen Shakki -lehden tehtäväshakkitoimittaja Kari Valtonen kommentoi tehtävääni mm. seuraavasti: “Uskoisin tehtäväsi ratkaistuani ihailevani enemmän laatijan juonikkuutta kuin omaa ratkaisunopeuttani, mikä on kiinnostavan tehtävän merkki!” Tehtävää ei kuitenkaan voitu julkaista tässä muodossa, koska se sisälsi edellisessä versiossa mainitun dualismin, jonka korjaaminen vaikutti pitkään mahdottomalta.

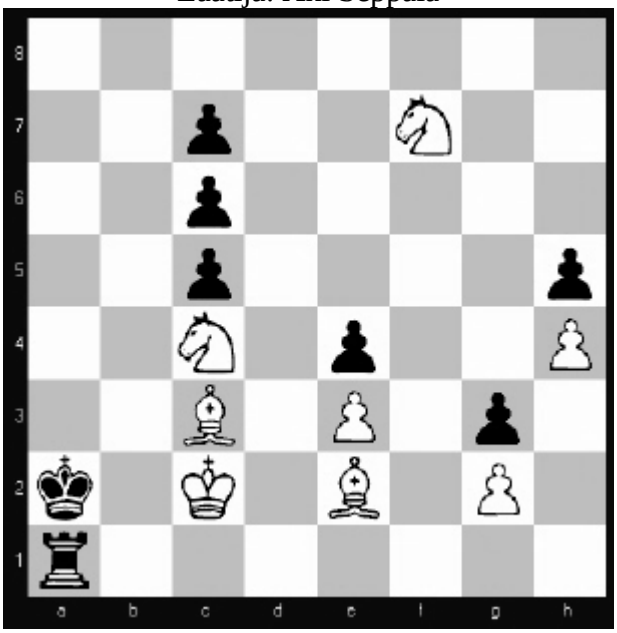
Ratkaisu: Houkutukset: 1.Lxh5? Txb1 2.Lf7+ Ka3 3.Kxb1 Ka4 4.Kc2 Ka3 5.Ld2 Ka4 6.Kc3 Ka3 7.Lc1+ Ka4 8.Lb3X

1.Rd2? Txd1 2.Kxd1 Ka3 3.Kc2 Ka2?(Ka4!) 4.g4 hxg4?(Ka3!) 5.Lb2 g3 6.Re4 g2 7.Rc3X

Oikein on: **1.Lxa1! Kxa1 2.Lg4!!** (2.g4? hxg4 3.h5 g3 4.h6 g2 5.h7 g1D 6.h8D+

Dg7!) **Ka2 (2. - hxg4 3.h5 Ka2 4.h6 Ka1 5.h7 Ka2 6.h8T! Ka1 7.Ta8X) 3.Le6+ Ka1 4.g4! hxg4 5.Kc1 g3 6.Ra3 g2 7.Rc2X**

Laatija: Aki Seppälä



7X 8 + 8

Selostus: Tämä on tehtävän lopullinen versio, joka siis julkaistiin Suomen Shakissa ja jonka ratkaisemiseen haastoin myös tämän tutkielman lukijan luvussa 5.3.2. Tehtävä sisältää suurimman osan edellisten versioiden hienouksista. Uutena elementtinä on huippuluokan houkutus 1.Rd2?, jossa ideana on 2.Lb2 3.Lc4X ja jonka musta voi kumota vain nipin napin siirrolla 1. - Ka3!!. Tämän siirron jälkeenkin näyttäisi, että valkea pystyy helposti tekemään matin neljällä upseerillaan 7. siirtoon mennessä, mutta musta pystyy viivyttämään tämän 8. siirtoon käyttämällä kaikki c-linjalla olevien sotilaiden viivytys-siirrot. Näiden sotilaiden ensisijainen tarkoitus oli korjata dualismiongelma, mutta nämä sotilaat toimivat upeasti myös tässä houkutusjatkossa. Toinen merkittävä houkutus on syödä torni ja yrittää tehdä matti tuomalla ratsu apuun. Ratsu on lisätty ensisijaisesti sen takia, että se estää sivuratkaisun 1.Lxh5, mutta se luo myös tämän houkutuksen ja lisäksi saa ensimmäisenä mainitun syvällisemmän houkutuksen sivuhoukutusmuunnelman (1.Rd2? Te1?) toimimaan täydellisesti.

Ratkaisu: Päähoukutus: 1.Rd2? Te1? (1. - Tg1?? 2.Lb2 3.Lc4X) 2.Lc4+! (2.La6? Ka3!) Ka3 3.Lb3! Tc1+ (3.- Te2 4.Re5! Txd2+ 5.Lxd2 c4 6.Rxc4X) 4.Kxc1 c4 5.Lc2! Ka2 (5. - c5 6.Kb1 c6 7.Rxc4X) 6.Lb2! c5/c3 7.Lb1X

Päähoukutuksen kumous: 1.Rd2? Ka3!! 2.Lxa1 (2.Lc4? Ka4! ja musta kuningas pakenee mattiverkosta ruutujen a5 ja b6 kautta, kaikki muut siirrot, kuten tornilla shakkaaminen, johtaisivat mattiin viimeistään 7. siirrossa) Ka4! (tai 2. - c4! 3.Lc3 c5! 4.Lxc4 Ka4! 5.Re5 c6! 6.Rxe4 Ka3 7.Lb3 c4 8.Rxc4X, mutta muut siirrot mahdollistavat matin ennen 8.siirtoa: 2. - Kb4? 3.Lc3+ Ka3 4.Rb3 Ka4 5.Kb2 c4 6.Lxc4 c5 7.Rxc5X; 2. - Kb2?? 3.Lb2 c4 4.Lxc4X) 3.Lc3 c4! (3. - Ka3? 4.Lc4 Ka4 5.Re5 Ka3 6.Lb3 c4 7.Rxc4X) 4.Lxc4 c5! (4. - Ka3? 5.Lb3 c5 6.Rc4X) 5.Re5 c6! (5. - Ka3? 6.Lb3 ja siirron matti) 6.Rxe4 Ka3 7.Lb3 c4 8.Rxc4X

Ei toimi myöskään 1.Ra5/Re5? Tg1! (1. - Ka3? ei auta tässä: 2.Lxa1 Ka4 3.Rxc6! c4 4.Lxc4 Ka3 5.Rd6!! cxd6 6.Lb3 d5 7.Lb2X; 1. - Tb1? 2.Lc4+ Ka3 3.Kxb1 Ka4 4.Kc2 Ka3 5.Lb3 c4 5.Rc4X; 1. - Th1?? 2.Lb2 3.Lc4X) 2.Lb2? Txg2! mustan eduin(!).

Oikea jatko on **1.Lxa1! Kxa1 2.Lg4!!** (2.Lxh5? Ka2 3.Lg6 Ka1 4.h5 ja on helppo laskea, että sotilas korottuu siirron liian myöhään, lupaavampi yritys on 2.Lxh5/Ld1/Lf1 Ka2 3.Rg5 Ka1 4.Rxe4 Ka2 5.Rc3+ Ka1 6.Ra3 c4 7.Kc1 c5 8.Rc2X, matti tuli siirron liian myöhään ja tässä voisi tulla mieleen, että mitäs jos siirtääkin heti ratsua niin mattikin tulee siirron nopeammin, mutta eipä tule patti pelastaa: 2.Rg5? Ka2 3.Rxe4 Ka1 ja patin takia nyt ei käykään 4.Rc3, vaan on pakko tehdä odotussiirto, ja ollaan takaisin edellisessä muunnelmassa, ja muutkaan tavat eivät tee mattia ennen 8.siirtoa, esim. 4.Rd2 Ka2 5.Ld3 Ka1 6.Kc1 Ka2 7.Lb1+ Ka2 8.Rb3X. voisi myös yrittää kierrättää ratsua apuun toista reittiä, mutta sekin on siirron verran liian hidas: 2.Re5? Ka2 3.Rxc6 Ka1 4.Kb3 Kb1 5.Rd2+ Kc1 (5. - Ka1 6.Ka3 c4 7.Rb4 c3 8.Rc2X) 6.Kc3 c4 7.Rb4 c5 8.Ra2X) **2. - Ka2 (2. - hxg4 3.h5 Ka2 4.h6 Ka1 5.h7 Ka2 6.h8T!** (6.h8D? patti) **Ka1 7.Ta8X) 3.Le6! Ka1 4.Ra3! c4 5.Lxc4!** (5.Kc1? Ka2!) **6.Kc1 c6 7.Rc2X**