Shakkiohjelmien kehitys heuristisesta hausta neuroverkkopohjaiseen itseoppimiseen

Tietotekniikka
Tietotekniikan laitos, Teknillinen tiedekunta
TkK-tutkielma
Tobias Santonen
Joulukuu 2024

Turun yliopiston laatujärjestelmän mukaisesti tämän julkaisun alkuperäisyys on tarkastettu
Turnitin OriginalityCheck -järjestelmällä.

TURUN YLIOPISTO

Tietotekniikan laitos

TOBIAS SANTONEN: Shakkiohjelmien kehitys heuristisesta hausta neuroverkkopohjaiseen itseoppimiseen

TkK-Tutkielma, 24 s.

Tietotekniikka

Joulukuu 2024

Shakki on pitkään ollut sovelluskohde tekoälylle. Tässä tutkielmassa tarkastellaan shakkiohjelmien erilaisia menetelmiä ja tekniikoita 1950-luvulta nykypäivään. Tutkielma keskittyy erityisesti alfa-beeta-karsintaan pohjautuviin hakumenetelmiin, mitkä ovat olleet ylivoimaisesti menestyneimmät tavat toteuttaa shakkia pelaava tietokoneohjelma.

Tietokonelaitteiston laskennallinen kehitys on mahdollistanut neuroverkkojen yleistymisen ja niiden hyödyntäminen shakin pelaamisessa on noussut tutkimuskohteeksi. Merkittävin neuroverkkoon pohjautuva shakkiohjelma AlphaZero osoitti perinteisesti kuvankäsittelyyn sovelletun konvoluutioneuroverkon soveltuvan myös shakin pelaamiseen. NNUE puolestaan osoitti, että perinteisestä neuroverkkorakenteesta voidaan kehittää tehokas shakin arviointimenetelmä.

Vaikka shakkiohjelmien kyky pelata shakkia on jo kauan sitten ylittänyt ihmispelaajien taidot, shakissa ei ole löydetty strategiaa, jota noudattamalla voitaisiin taata pelin lopputulos.

Asiasanat**:** Shakki, peliteoria, lautapeli, tekoäly, alpha-beta-karsinta, neuroverkko

Sisällys

1 Johdanto 1

2 Shakin luonne ja shakkiohjelmien historiaa 3

3 Shakkiohjelmien käyttämiä tekniikoita ja algoritmeja 6

3.1 Arviointifunktio 6

3.2 Haku shakkiohjelmissa 7

3.3 Minimax 8

3.4 Minimax-algoritmin optimointimenetelmät 10

3.4.1 Tappaja- ja historiaheuristiikka 12

3.4.2 Nollasiirtokarsinta 12

3.4.3 Transpositiotaulu 12

3.4.4 Päävariaatiohaku 13

3.5 NNUE: Neuroverkko arviointifunktiona 14

3.6 Monte carlo –puuhaku 16

4 AlphaZero ja LeelaChessZero 18

5 Pohdintaa 21

6 Yhteenveto 23

Lähteet 25

1 Johdanto

Shakki on lautapeli, jonka juuret ulottuvat yli tuhannen vuoden taakse. Nykyiset shakin säännöt vakiintuivat 1600-luvun loppuun mennessä.[1] Ensimmäinen virallinen shakkiturnaus pidettiin Lontoossa vuonna 1851, mikä oli tärkeä merkkipaalu pelin historiassa ja asetti kilpailulliselle pelaamiselle yhtenäiset säännöt.

Monet pelit, kuten esimerkiksi neljän suora ja ristinolla, ovat ratkaistuja. Neljän suorassa aloittava pelaaja voittaa aina optimaalista strategiaa käyttäen, kun taas ristinolla päätyy aina tasapeliin, jos molemmat pelaajat pelaavat optimaalisesti. Shakki on toistaisesti ratkaisematon ja Claude Shannon arvioi vuonna 1950 shakin pelipuun kompleksisuuden alarajaksi 10^{120}. [2] Tämä luku on huomattavasti suurempi kuin kaikkien atomien yhteenlaskettu määrä maailmankaikkeudessa, joten shakin ratkaiseminen käymällä läpi jokainen mahdollinen pelin kulku on käytännössä mahdotonta nykyteknologialla.

Vuonna 1956 MANIAC I -tietokone oli ensimmäinen tietokone, joka voitti ihmispelaajan shakissa. Tämä shakkiottelu käytiin variantissa, jossa pelilauta oli 6x6 normaalin 8x8 sijaan ja lähetti puuttui, mikä johtui tietokoneen rajoitetusta muistista ja prosessointikyvystä. [3] Vuoteen 1997 mennessä tietokoneiden prosessointikyky oli kehittynyt huomattavasti ja Deep Blue -tietokone voitti shakin hallitsevan maailmanmestarin Kasparovin. Shakkitietokoneet ja tekoäly ovat luonnollinen jatkumo shakin kilpailulliselle pelaamiselle, sillä Deep Bluen jälkeen shakkiohjelmia tai shakkitietokoneita voidaan pitää parhaimpina shakin pelaajina.

Tekoäly tieteenalana tutkii tietokonejärjestelmiä tai tietokoneohjelmia, jotka kykenevät ratkaisemaan ongelmia, jotka vaativat erikoistietoa tai taitoa. Tekoälyksi voidaan myös kutsua ohjelmia, jotka pyrkivät jollain tapaa imitoimaan ihmisen aisteja tai muistia. Varhaiset shakkiohjelmat perustuivat sääntöpohjaisiin algoritmeihin, jotka analysoivat pelitilanteita ihmisten antamien sääntöjen mukaisesti. Näiden ohjelmien vahvuus oli niiden kyky käydä läpi suuria määriä siirtovaihtoehtoja ja niiden heikkoutena oli puutteellinen ymmärrys shakista.

Koneoppiminen on tekoälyn osa-alue, joka tutkii algoritmeja, jotka kykenevät oppimaan datasta piirteitä ja ominaisuuksia. Syväoppimiseksi kutsutaan koneoppimisen menetelmiä, jotka hyödyntävät neuroverkkoja. Neuroverkot ovat ihmisaivojen toiminnasta inspiraationsa saaneita malleja, jotka koostuvat keinotekoisista neuroneista. Neuroverkkojen menestys kuvantunnistuksessa ja luonnollisen kielen käsittelyssä on tehnyt syväoppimisesta suositun tutkimuskohteen. Kuvantunnistukseen soveltuva neuroverkkorakenne on soveltunut myös

shakkiin. Neuroverkkojen etu shakissa aikaisempiin ohjelmiin verrattuna on se, että se kykenee oppimaan taktisia kuvioita tai shakkistrategioita, mitkä eivät ole ihmisten opettamia. Tällöin on mahdollista, että ohjelman ymmärrys shakista voi kasvaa laajemmaksi kuin ihmisen.

Tässä tutkielmassa on tarkoituksena vastata seuraaviin tutkimuskysymyksiin:

1. Minkälaisia tekniikoita ja algoritmeja shakkitekoäly hyödyntää?

2. Miten neuroverkkoja hyödynnetään shakkiohjelmissa?

Tämä tutkielma on kirjallisuuskatsaus ja lähteet on pääosin haettu ACM Digital Librarysta, Google Scholarista, arXivista. Lähteinä on myös käytetty nykyaikaisten shakkiohjelmien Stockfishin ja LeelaChessZeron lähdekoodia. Shakki on herättänyt kiinnostusta tekoälyn sovelluskohteena jo 1950-luvulta saakka, joten kirjallisuutta löytyy monilta vuosikymmeniltä. Hakutermeinä on käytetty “Chess Engine”, “Chess AI”, “Computer chess” ja lisäksi, koska useat tekniikat ja menetelmät ovat käyttökelpoisia myös muissa lautapeleissä, on hakuun käytetty myös “Computer Shogi” ja “Computer Go”.
Tutkielma koostuu neljästä luvusta. Luvussa kaksi esitellään shakin sääntöjä ja erilaisia ominaisuuksia, jotka ovat tärkeitä tekoälyä luotaessa. Lisäksi luvussa kaksi esitellään shakkiohjelmien historiaa. Kolmannessa luvussa käydään läpi tärkeimpien menetelmien toteutuksia. Neljäs luku sisältää tutkielmasta pohdintaa ja yhteenvedon tutkimuksesta.

2 Shakin luonne ja shakkiohjelmien historiaa

Shakki on nollasummapeli, jossa tavoitteena on matittaa vastustajan kuningas. Matitus tapahtuu, kun vastustajan kuningas on uhattuna eikä voi paeta vangitsemista. Kumpikin pelaaja hallitsee yhteensä 16:ta shakkinappulaa, jotka ovat kuningas, kuningatar, kaksi tornia, kaksi ratsua, kaksi lähettiä ja kahdeksan sotilasta.

Vaikka kuninkaan matittaminen on ainoa tapa voittaa, shakkipelin voitto on usein saavutettu jo ennen mattia. Monissa tilanteissa materiaalietu tai aktiivinen nappuloiden asemointi johtaa etuihin, jotka kasvavat, kunnes lopulta matitus on triviaalia. Shakille on ominaista myös saavutetun edun peruuttamattomuus. Kun pelaaja saavuttaa edun, esimerkiksi materiaalin tai paremman aseman muodossa, etua on vaikea menettää. Tämä tekee tilanteiden arvioinnista erityisen tärkeää, sillä arviointivirheet johtavat korjaamattomiin tappioasemiin. Erilaisten pelitilojen arviointi, eli se kummalla pelaajalla on etu, on hankalaa esimerkiksi siksi, että materiaalitappiota voi kompensoida shakkinappuloiden paremmat asemat. Jotta voidaan löytää tilanteessa paras liike, on oltava jokin menetelmä arvioida pelitiloja, johon liike johtaa. Kleinin mukaan arvioitavia asioita ovat sotilasrakenne, vahvat ja heikot ruudut, avoimet linjat ja kuninkaan turvallisuus [4]. Deep Bluen aseman arviointi otti huomioon jopa 8000 erilaista piirrettä tai ominaisuutta, jota shakissa voi esiintyä [5].

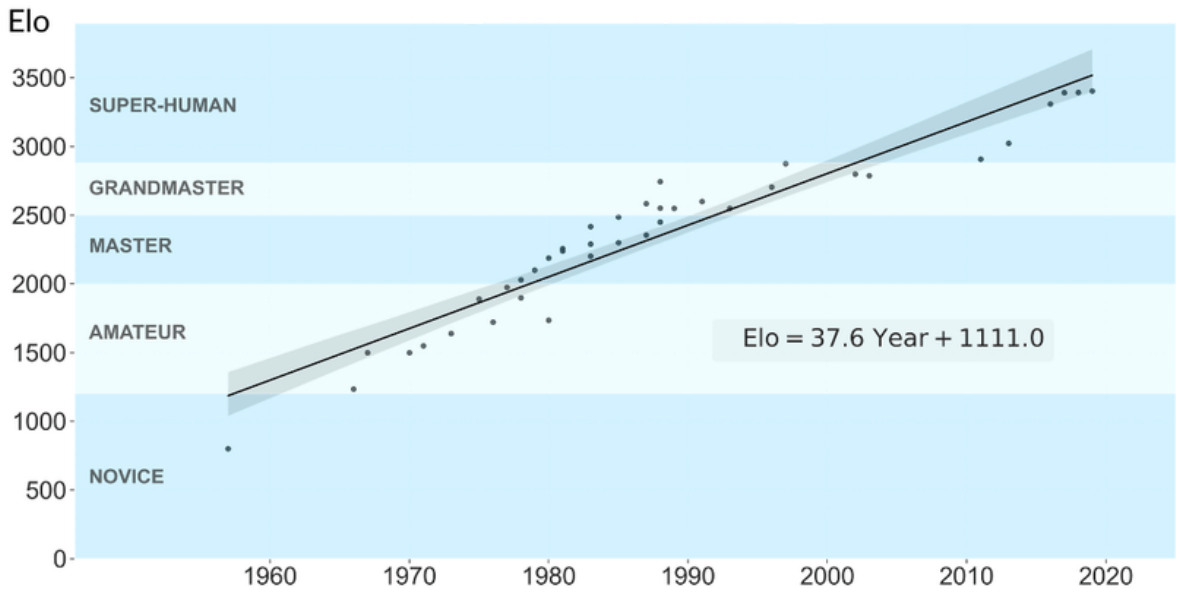
Pelin kulku on perinteisesti jaettu kolmeen osaan: avaus-, keski- ja loppupeliin. Useat avaukset on hiottu vuosisatojen saatossa ja pelaajat ovat tutkineet niiden variaatioita, heikkouksia ja vahvuuksia. Historiallisesti shakkiohjelmaan on kuulunut avauskirja, josta shakkiohjelma ottaa avauspelissä tapahtuvat siirrot suoraan muistista haun sijaan. Modernit neuroverkkopohjaiset tekoälyt, kuten AlphaZero ja LC0, ovat kuitenkin osoittaneet kykenevänsä pelaamaan vahvaa shakkia myös ilman ennalta määriteltyä avauskirjastoa [4]. Sen sijaan useat loppupelit ovat matemaattisesti ratkaistuja, jolloin peli on pakotettu tietylle tielle, joka päätyy voittoon tai tasapeliin. Tätä varten on luotu erillisiä loppupelitietokantoja, joissa on valmiiksi laskettu täydellinen pelitapa tietyille nappulamäärille. Jäljelle jäävä keskipeli on siten mielenkiintoisin, koska se on usein täysin ratkaisematon ja vaatii monimutkaista strategista ja taktista ymmärrystä.

Nimi	**Tekijöiden nimet**	**Luotu**	**Algoritmit, metodit ja käytetty teknologia**
Turochamp	Alan Turing, David Champernowne	1948	Muuttuva ennakointi, kahden siirron heuristiikka, arviointifunktio
NSS	Allen Newell, Herbert Simon, Cliff Shaw	1958	Arviointifunktio, α-β haku
Mac Hack VI	Richard Greenblatt	1966	Arviointifunktio, α-β haku, transpositiotaulu
First Macedonian chess program	Stevo Božinovski	1969	Arviointifunktio
Chess (Northwestern University Chess 4.5)	Larry Atkin, David Slate	1975	Arviointifunktio, α-β haku, transpositiotaulu, bittilauta, täysleveä haku, iterativiinen syventäminen
Belle	Ken Thompson, Joe Condon	1976	Siirronmuodostaja, arviointifunktio, α-β haku, transpositiotaulu, Laiska ja täysi arviointi, Päävarianttien jakaminen
HiTech	Hans Berliner, Carl Ebeling, Murray Campbell, Gordon Goetsch	1985	Arviointifunktio, kaavojen tunnistus, transpositiotaulukko, rinnakkainen haku ja α-β haku
Fritz	Frans Morsch, Mathias Feist	1991	Arviointifunktio, rinnakkainen haku ja nollasiirtohaku
Deep Blue	IBM Development Team	1996	Arviointifunktio, α-β haku,transpositiotaulu, rinnakkainen haku ja yksittäinen laajennus.
Stockfish	Tord Romstad, Marco Costalba, Joona Kiiski, Gary Linscott	2016	Arviointifunktio rinnakkainen haku, transpositiotaulu, arvokkaimman uhrin/vähiten arvokkaan hyökkääjän siirto, nollasiirron haku, yksittäiset laajennukset, turhuusharvennus
AlphaZero	DeepMind Development Team	2017	Neuroverkot, syväoppiminen, vahvistusoppiminen, Monte Carlo - puuhaku, siirtotranspositiotaulu
NNUE	Yu Nasu	2020	Neuroverkko arviointifunktiona

Taulukko 1 Historialliset shakkiohjelmat ja niiden käyttämiä menetelmiä. Pääosin lähteestä [6]

Taulukossa 1 on esitetty merkittäviä shakkiohjelmia ja shakkitietokoneita, joista kaikki paitsi AlphaZero ovat alpha-beta hakijoita. Muita alpha-beta hakua huonommin menestyneitä hakumenetelmiä ovat muun muassa B*-algoritmi [7] ja SSS*-algoritmi [8].

Elojärjestelmä on shakissa käytettävä tilastollinen järjestelmä, joka pyrkii ilmaisemaan pelaajan taitotason muihin pelaajiin verrattuna. Kahden pelaajan eloluvun erotus antaa odotusarvon niiden keskenään pelattavissa otteluissa. Esimerkiksi pelaajalla, jolla on 100 elopistettä suurempi eloluku kuin vastustajallaan, on 0,64 pisteen odotusarvo systeemissä, jossa voitto on 1 pisteen, tasapeli 0,5 pisteen ja häviö 0 pisteen arvoinen.



Kuva 1 Shakkiohjelmien kehitys [9]

Arvio shakkiohjelman taitotasosta voidaan saada peluuttamalla ohjelmia vastakkain. Shakkiohjelman vahvuus riippuu hyvin paljon siitä, kuinka paljon aikaa sille annetaan siirron hakuun. Shakkiohjelman vahvuus on tällöin riippuvainen sen käyttämästä laitteistosta, mikä selittänee kuvassa 1 esiintyvät poikkeavat arvot, jolloin samoja metodeja käyttävät ohjelmat saattoivat vuosien mittaan saada eri eloluvun.

Esimerkiksi tutkielmassa esitellyt Stockfish ja LC0 ovat erilaisia ohjelmia sillä tavalla, että LC0:n syvien neuroverkkojen takia se suoritetaan pääosin grafiikkasuorittimilla, kun taas Stockfish suorittimella. Tämän takia on vaikea sanoa, mikä olisi reilu laitteistokonfiguraatio. Tässä tutkielmassa luotetaan pitkälti Top Chess Engine Championshipin (TCEC) -turnausten tuloksiin. TCEC:ssä jokaiselle ohjelmalle annetaan sama laitteisto, johon on kuulunut vuodesta 2021 lähtien kaksi Xeon 6230R -CPU:ta ja kaksi A100 -GPU:ta.

3 Shakkiohjelmien käyttämiä tekniikoita ja algoritmeja

Tässä luvussa alaluvut 3.1–3.5 käsittelevät perinteisen shakkiohjelman oleellisimmat osat ja menetelmät. Alaluvussa 3.6 esitelty hakumenetelmä on sen sijaan otettu käyttöön shakkiohjelmissa vasta hiljattain. Menetelmää on hyödynnetty shakkiohjelmissa, joissa on käytetty laskennallisesti raskaan neuroverkon antamaa ohjausta, kuten luvussa 4 esitellyssä AlphaZerossa.

3.1 Arviointifunktio

Kaikille shakkiohjelmille olennainen kyky on arvioida pelitilannetta. Yksi tapa arvioida tilannetta on laskea kaikki pelissä olevat nappulat. Shannonin mukaan shakkinappuloiden suhteelliset arvot ovat sotilas = 1, ratsu = 3, lähetti = 3, torni = 5 ja kuningatar = 9. Tämä riittää hyvin yksinkertaisen arviointifunktio luontiin, jos lisätään vielä kuninkaan arvoksi joku suhteellisen suuri luku kuten kuningas = 1000. [2]

Kuva 2 Mustalla on materiaalietu, mutta mustan kuningas on avoin hyökkäykselle

Materiaaliin perustuva arviointi laskettaisiin kaavalla:

$$F(p,n,b,r,q,k,P,N,B,R,Q,K) = (p-P) + 3(n-N) + 3(b-B) + 5(r-R) + 9(q-Q) + 1000(k-K)$$

Funktio saa syötteeksi pelissä olevien nappuloiden lukumäärät, jossa p on valkoisen sotilaat, P on mustan sotilaat, n valkoisen ratsut, N mustan ratsut, b valkoisen lähetit, B mustan lähetit, q valkoisen kuningatar, Q mustan kuningatar, k valkoisen kuningas ja K mustan kuningas

Tällöin arvio kuvassa 2 olisi $F(7,2,2,2,0,1,7,2,1,2,1,1) = -6$

Kun funktion arvo on positiivinen, valkoinen on voitolla ja jos se on negatiivinen, musta on voitolla. Jos arvo on 0, niin peli on tasan. Koska kuvan 2 tilanne on hiljainen, yksinkertainenkin arviointifunktio antaa kohtuullisen hyvä arvion; Stockfish arvioi syvyydellä 40 tilanteen olevan –5,0.

Materiaalin lisäksi voidaan arvioida lukuisia asioita, joita voidaan ilmaista matemaattisesti. Tämän tyylisiä arviointifunktioita kutsutaan käsintehdyiksi arviointifunktioiksi (engl. handcrafted evaluation function). Käsintehdyn arviointifunktion hyvyys rajoittuu sen kehittäjien shakkiymmärrykseen. On myös tilanteita, jotka ihminen intuitiivisesti arvioi hyviksi tai huonoiksi. Tunnetta kuten “Tämä tilanne tuntuu huonolta” on vaikea ilmaista matemaattisena lausekkeena. Joitakin epälineaarisia suhteita on myös vaikea muodostaa matemaattisesti. Mitä monimutkaisempi arviointifunktio, sitä enemmän se käyttää aikaa tilanteiden arvioimiseen, joka vie aikaa haulta.

3.2 Haku shakkiohjelmissa

Shakkiohjelmalle olennainen taito on navigointi monimutkaisessa päätöksentekotilassa, jossa jokainen oma siirto johtaa useisiin vastustajan mahdollisiin siirtoihin, jotka puolestaan johtavat vastauksiin. Tämä dynamiikka luo laajan pelipuun, joka kasvaa eksponentiaalisesti: Shannon arvioi shakissa olevan keskimäärin noin 30 sallittua siirtoa [2] vuorossa. Tällöin esimerkiksi kymmenen siirron jälkeen pelipuun leveys olisi noin 30^{10} solmua. Tämän vuoksi tarvitaan erilaisia menetelmiä pelipuun tehokkaaseen läpikäyntiin. Jokainen merkittävä shakkiohjelma käyttää jossakin toiminnan vaiheessa jonkinlaista hakua [9].

Haun toteuttamiseksi shakki täytyy mallintaa muotoon, jota tietokone pystyy käsittelemään. Shakin nappuloilla on monta erilaista erikoissääntöä ja niiden huomioon ottaminen on edellytys, jotta voidaan tehdä haku. Pitää myös ottaa huomioon se, että shakkipelin edetessä tietyt erikoissiirrot, kuten linnoitus, eivät välttämättä ole sallittuja siirtoja. Deep Bluen arkkitehti ja pääsuunnittelija Feng-Hsiung Hsun mukaan Deep Blue kärsi ”Ghost Queen” -viasta, missä Deep Blue arvioi pelitilanteen täysin väärin, koska pelilaudalle ilmestyi

kuningatar, joka ei todellisuudessa ollut pelissä [5]. Tämänkaltaisten ongelmien takia on välttämätöntä, että haun kaikki siirrot ovat sallittuja ja shakin pelitila on esitetty virheettömästi.

Sallittujen siirtojen muodostamisesta vastaa siirronmuodostaja (engl. move generator). Pelitilanne, eli pelissä olevat nappulat ja niiden sijainti, voidaan esittää esimerkiksi bittilautana [10]. Bittilauta hyödyntää sitä, että shakin pelilaudalla on 64 ruutua, jotka voidaan esittää yhtenä 64-bittisenä kokonaislukuna, jonka jokainen bitti edustaa yhtä pelilaudan ruutua. Esimerkiksi jos ruutua A1 vastaa kokonaisluvun ensimmäinen bitti ja ruudulla A1 sijaitsee nappula, bitin arvo on 1 ja jos ruutu on vapaa, bitin arvo on 0. Näin ei kuitenkaan vielä saada eroteltua kaikkia nappuloita, joten bittilautaa varten tarvitaan kaksitoista 64-bittistä kokonaislukua, joista jokainen vastaa yhtä tietyn tyyppistä nappulaa.

3.3 Minimax

Minimax-algoritmi on strategia, jota käytetään kaksinpeleissä, kuten shakissa, valikoimaan optimaalisia siirtoja arvioimalla mahdollisia pelitiloja. Minimax hyödyntää puurakennetta kuvaamaan kaikkia mahdollisia siirtoja ja niistä seuraavia pelitiloja. Puun jokainen solmu edustaa pelitilaa ja jokainen puun haara pelaajan siirtoa. Minimax olettaa, että molemmat pelaajat pelaavat optimaalisesti: pelaaja, jonka vuoro on, pyrkii maksimoimaan arviointifunktion arvoa, kun taas vastustaja pyrkii minimoimaan sitä. [4]

Minimax-algoritmi voidaan muodostaa seuraavasti: valkoisen vuorolla muodostetaan puu kaikista mahdollisista siirroista, joita valkoinen voi tehdä. Näille siirroille muodostetaan kaikki mustan mahdolliset vastasiirrot. Tätä prosessia jatketaan, kunnes saavutetaan haluttu syvyys tai kunnes peli päättyy tai aika ja muistitila loppuvat kesken. Tässä vaiheessa jokainen lehtisolmu eli solmu, jolla ei ole lapsia, arvioidaan käyttäen arviointifunktiota. Lopulta tulokset välitetään askel askeleelta lehtisolmulta takaisin juurisolmulle, siten että jokainen solmu, jonka lapsisolmu on arvioitu, saa arvokseen suurimman lapsisolmunsa arvon, jos on valkoisen vuoro ja pienimmän lapsisolmunsa arvon, jos on mustan vuoro. Pelitilanteessa seuraavaksi siirroksi valitaan juurisolmun lapsisolmuista se, jolla on suurin arvo, jos on valkoisen vuoro ja pienimmän arvon lapsisolmu, jos on mustan vuoro. [4]

Kuva 3 Taktisesti epävakaa pelitilanne

Tarkastellaan kuvan 3 tilannetta. Valkoinen pelaaja on lyönyt mustan lähetin kuningattarellaan, minkä seurauksena valkoisella tilapäisesti on yhden lähetin materiaalietu. Kuitenkin mustalla on useita vaihtoehtoja lyödä valkoisen kuningatar, jonka jälkeen yksinkertainen materiaaliin perustuva arviointi antaisi mustalle vahvan edun. Kutsutaan tätä tilannetta, jossa valkoisella on lähettietu, vuoroksi n. Jos hakusyvyys olisi asetettu siten, että haku päättyy vuoroon n, menisi arviointi pieleen, sillä se ei huomioisi mustan ilmeistä vastausta. Tämän kaltaisen ongelman ratkaisee hiljaisuushaku.

Hiljaisuushaun tarkoituksena on jatkaa hakua tilanteissa, joissa on käynnissä taktinen uhka tai epävakaus [11]. Hiljaisuushaku on keskeinen osa hakuprosessia ja se mainittiin ensimmäisen kerran jo 1950-luvulla [2]. Yksinkertainen hiljaisuushaku voidaan toteuttaa tarkastamalla tilanteessa olevia mahdollisia lyöntejä. Jos tilanteessa ei ole mahdollisia lyöntejä, haku voidaan lopettaa. Jos tilanteessa on lyöntejä, niistä muodostetaan pelipuu, eli lyöntipuu, jonka avulla haetaan ja arvioidaan ne pelitilanteet, joihin eri lyönnit johtavat [11].

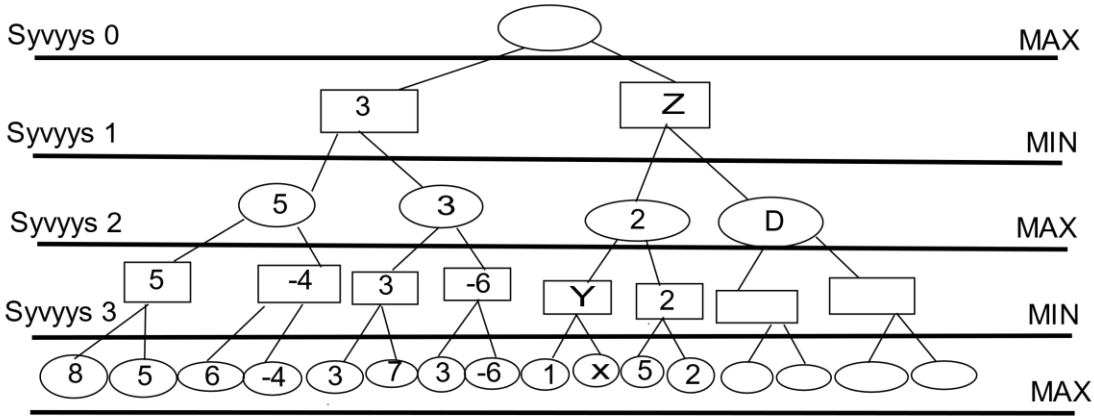
Nollasiirroksi (engl. null move) kutsutaan siirtoa, jossa pelaaja ei tee siirtoa vuorollaan vaan antaa vuoron toiselle pelaajalle. Vaikka nollasiirto ei ole sallittu shakissa, sitä voidaan käyttää

arvioimaan tilanteen hiljaisuutta. Jos pelaaja tekee nollasiiron ja se ei vaikuta tilanteen arvioon merkittävästi, tilannetta voidaan pitää hiljaisena. [11]

Shakkia pelatessa yleensä pelille asetetaan aikaraja. Pelaajat saavat itse päättää kuinka paljon omaa aikaa käyttää tietyllä vuorolla. Ennen hakua ei voida tietää kuinka kauan siirron etsimisessä kestää, jos se suoritetaan tiettyyn syvyyteen. Sen sijaan, että haetaan syvyyteen N, haku jaetaan siten, että aloitetaan haulla syvyydelle 1, jonka jälkeen tehdään haku syyvydelle 2. Tätä jatketaan syvyyskerroksittain, kunnes löydetään pelin loppu tai kunnes haulta loppuu käytettävä aika. Tätä kutsutaan iteratiiviseksi syventämiseksi (engl. iterative deepening). [12] Huomion arvoista on se, että iteratiivisessa syventämisessä tehdään ylimääräistä ja toistettua laskentaa. Tämän ylimääräisen laskennan merkitystä vähentää se, että suurin osa laskenta-ajasta käytetään syvimmän syvyystason arvioimiseen. Iteratiivisen syventämisen aikavaatimus on siis täten huonoimmillaan siltikin sama kuin naiivin minimaxin. [4]

3.4 Minimax-algoritmin optimointimenetelmät

Minimaxin aikavaatimus on $O(b^d)$, jossa b on hakusyyvys ja d on haarautumiskerroin. Sen keskeinen optimointimenetelmä on alpha-beta-karsinta, jonka aikavaatimus on parhaimmillaan $O(\sqrt{b^d})$. Koska alpha-beta-karsinta on huonoimmassa tapauksessa yhtenäinen minimaxin kanssa, ei ole syytä olla käyttämättä sitä, joten shakkiohjelmia on historiallisesti kutsuttu alpha-beta-hakijoiksi. Alpha-beta-hakuun on kehitetty paljon erilaisia menetelmiä, jotka eivät välttämättä ole keskenään yhteensopivia ja niiden tuottamat hyödyt ovat välillä vaikeasti todistettavia. Esimerkiksi tappajaheuristiikan (ks. sivu 12) on toisinaan havaittu vähentävän jopa tutkittavien solmujen lukumäärää 80% ja toisinaan vähennys on ollut merkityksetön. [12]



Kuva 4 Yksinkertainen pelipuu

Tarkastellaan kuvassa 4 olevaa pelipuuta. Puun solmuja on arvioitu vasemmalta oikealle ja arviointi on kesken tilanteessa, jossa solmujen arvot ovat 1 ja X ja lisäksi tiedetään, että vuoro on minimoivalla pelaajalla. Eli solmun Y arvo on $\min(1, X)$. Koska tiedetään, että syvyydellä 1 maksimoivalla pelaajalla on jo siirto, jonka arvo on 3 ja $3 > \min(1, X)$, X :n arvosta riippumatta maksimoiva pelaaja tulee valikoimaan arvon 3. Tämän seurauksena X -solmua ei tarvitse arvioida ja se voidaan karsia. Seuraavaksi voidaan arvioida $Z \leq 2 < 3$, joten koko oikea alipuu Z -solmusta alkaen voidaan karsia. Alpha-beta-karsintaa on käytetty shakkiohjelmissa 1950-luvulta lähtien [2]. Minimax ja alpha-beta-karsinta toimivat shakista riippumattomasti ja niitä käytetään myös esimerkiksi Shōgissa. [13]. Karsinta nopeuttaa hakua kahdella tapaa: karsittuja solmuja eli pelitilanteita ei tarvitse arvioida eikä niiden siirtoja tarvitse muodostaa siirronmuodostajalla.

Koska alpha-beta-karsinta tehdään syvyyssuuntaisesti, sitä voidaan parantaa heuristiikalla. Jos tiedetään tiettyjen siirtojen olevan jollain tapaa lupaavia, ne voidaan järjestää vasemmalta oikealle, parhaimmasta huonoimpaan. Mitä paremmin tämä siirtojärjestys toteutuu, sitä enemmän karsitaan solmuja. Esimerkiksi siirtoja, joissa sotilas voi lyödä arvokkaamman nappulan, kuten tornin, voidaan yleisesti pitää hyvinä siirtoina. Alpha-solmuksi kutsutaan maksivoivan pelaajan parasta solmua ja Beta-solmuksi kutsutaan minimoivan pelaajan parasta solmua. Näiden väliä (α, β) kutsutaan alpha-beta väliksi (engl. alpha-beta window). Solmu karsitaan, jos sen arvo on pienempi kuin α maksimoivan pelaajan vuorolla tai suurempi kuin β minimoivan pelaajan vuorolla. [4]

3.4.1 Tappaja- ja historiaheuristiikka

Siirtoja, jotka ovat jo joissain kohtaa hakua karsineet muita siirtoja, kutsutaan tappajasiirroiksi (engl. killer moves). Tappajaheuristiikka (engl. killer heuristics) on tapa järjestää siirrot puussa siten, että siirrot, jotka ovat jo tietyllä syvyydellä karsineet muita siirtoja, tutkitaan ensimmäisenä saman alipuun seuraavalla vuorolla. [14]

Historiaheuristiikka on tappajaheuristiikan yleistys koko pelipuulle. Sen sijaan, että järjestetään tietyn alipuun siirtoja, koko pelipuun siirtoja luokitellaan niiden aiheuttamien karsintojen perusteella. Siirtoja, jotka ovat tietyn syvyyden tai alipuun parhaimman arvon saaneita, voidaan myös arvioida paremmiksi. Näin voidaan luoda luokittelu, jossa siirolle annetaan historiallinen arvo (engl. history score). [12]

3.4.2 Nollasiirtokarsinta

Toinen tapa karsia solmuja on nollasiirtokarsinta (engl. null move pruning). Karsinta aloitetaan tekemällä nollasiirto, eli vastustaja saa tehdä kaksi siirtoa peräkkäin. Näiden kahden siirron jälkeen suoritetaan nopea pintahaku, joka on noin 3-5 siirtoa syvä. Kahden peräkkäisen siirron tekeminen oletetaan tuottavan edun shakissa. Jos pintahaun arvioinnin jälkeen kahden siirron tehnyt pelaaja ei ole saanut etua, voidaan alipuu alkaen solmusta, jossa nollasiirto tehtiin, karsia. Alipuun oletetaan olevan niin huono vastustajan pelaajalle, että vastustaja välttää sen. Näin ollen kyseisen alipuun karsisi myös alpha-beta-karsinta mutta vasta syvemmän haun jälkeen. Koska hakupuu kasvaa eksponentiaalisesti jokaisen siirron jälkeen, mitä aikaisemmin karsinta tapahtuu, sitä parempi. Shakissa on kuitenkin pelitilanteita, joita kutsutaan siirtopakoksi eli tilanteita, missä kaikki sallitut siirrot huonontavat peliasemaa. Ohjelma, joka hyödyntää nollasiirtokarsintaa voi karsia parhaimman siirron alipuun, jos siirtopakkoa ei tunnisteta. [4]

3.4.3 Transpositiotaulu

Shakissa transpositioksi kutsuaan tilannetta, jossa samaan peliasemaan johtaa kaksi eri siirtojen sarjaa. Tämän seurauksena pelipuussa on monta solmua, jotka vastaavat täysin samaa peliasemaa. Transpositiotauluksi kutsutaan memoisaatiotekniikkaa, jossa tietokoneen muistiin tallennetaan solmun peliasema, siihen liittyvä arviointifunktion antama arvio ja miltä syvyydeltä peliasema löytyi. Tällöin, kun sama pelitilanne kohdataan toiseen kertaan, voidaan arviointifunktion sijaan hakea tilanteen arvio muistista. [15]

Kymmenen siirron pelipuu on noin $\sum_{n=1}^{10} 30^n \sim 6{,}1 \cdot 10^{14}$ solmua. Tällöin kymmenen siirron pelipuun transpositiotaulu vaatisi bittilautaa käyttäen $6{,}1 \cdot 10^{14} \cdot 64 \cdot 12 \sim 5{,}9 \cdot$

10^{16} bittiä. Suuren muistin käytön vuoksi transpositiotaulut toteutetaan hajautusalgoritmilla. Yleisesti käytetty hajautusalgoritmi shakkiohjelmissa on Zobrist. [15]

Suuren solmujen lukumäärän seurauksena on todennäköistä, että eri peliasemat päätyvät samaan taulukon indeksiin. Tätä kutsutaan hajausavainten törmäykseksi (engl. hash collision) Tällöin tarvitaan jonkinlainen korvausmenetelmä (engl. heplacement scheme). Korvausmenetelmää tarvitaan myös silloin, kun hajautustaululta loppuu muistitila. Transpositiot tapahtuvat yleensä suhteellisen lokaalisti verrattuna globaaliin pelipuuhun, joten yksi oleellinen tapa on korvata mikä tahansa aikaisempi solmu viimeisimmällä, sillä viimeisin arvioitu solmu on läheisin haun nykyiseen tilaan. Toinen tapa on syvyyskorvaus, joka priorisoi solmuja syvyyden perusteella. Syvemmällä arvioituja solmuja pidetään arvokkaampina siksi, että niiden arvioimiseen on käytetty enemmän aikaa, joten niiden ollessa transpositioita, ne myös säästävät enemmän aikaa. Syvemmällä tutkitut solmut ovat myös tarkempia arvioita pelistä. [15]

3.4.4 Päävariaatiohaku

Minimaxin perusoletuksena on, että molemmat pelaajat pelaavat optimaalisesti. Shakissa molemmat pelaajat saavat identtiset pelinappulat ja aloituspeliaseman. Tällöin ainoa ero pelaajien välillä on se, että valkoinen saa edun, koska se saa tehdä ensimmäisen siirron. Alphan ja betan ero optimaalisessa pelipuussa on $\alpha - \beta$ = W, missä W on ensimmäisen siirron tuoma etu. Shakin korkeimmalla elotasolla ylivoimaisesti yleisin pelin lopputulos on tasapeli, joten voidaan olettaa että W on hyvin pieni. Tätä ominaisuutta voidaan hyödyntää karsimaan alipuita, joiden alpha-beta välit ovat suuria. [16]

Päävariaatiohaun tarkoitus on järjestää siirtoja iteratiivista syventämistä käyttäen siten, että haun jokainen syvyyskerros aloitetaan tutkimalla jokaisen syvyystason paras solmu. Jokaisen syvyyskerroksen parhaiden solmujen sarjaa kutsutaan päävariaatioksi ja se tutkitaan käyttäen kokonaista alpha-beta väliä. Kun on tutkittu syvyyskerrokseen N saakka, oletetaan, että päävariaatio säilyy edelleen parhaana siirtojen sarjana syvyyskerrokselle $N + 1$. Haku päävariaation ulkopuolella toteutetaan käyttäen minimaalista väliä eli $(\alpha, \alpha + 1)$. Jos päävariaation ulkopuolisen solmun arvo on pienempi kuin α , se voidaan karsia koska se on huonompi siirto kuin päävariaation siirto. Jos siirto on parempi kuin $\alpha + 1$ eli β , se voidaan karsia koska sen oletetaan olevan liian hyvä siirto toiselle pelaajalle. Jos todetaan, että siirto

on parempi kuin α , koko haku toteutetaan uudelleen käyttäen kokonaista alpha-beta väliä, jotta saadaan siirron todellinen arvo. [16]

3.5 NNUE: Neuroverkko arviointifunktiona

NNUE eli tehokkaasti päivittyvä neuroverkko on nykyaikaisten shakkiohjelmien käyttämä neuroverkko, joka korvaa käsintehdyn arviointifunktion. Shōgi on shakin kaltainen lautapeli, jota pelataan Japanissa. Shakin ja shōgin yhteisten piirteiden seurauksena useat tekniikat ja algoritmit toimivat molemmissa peleissä. Yu Nasu esitti neuroverkon, jonka hän nimesi NNUE:ksi (engl. *Efficiently Updatable Neural Networks*) Motohiro Isozaki, joka loi ensimmäisen alpha-beta-hakijan, joka hyödynsi NNUE:ta shōgissa, loi myös Stockfishille prototyyppi version NNUE:sta, joka arvioi shakkia shōgin sijaan. NNUE paransi Stockfishin elolukua lopulta noin 80 pisteen verran ja se on ollut NNUE:n lisäyksestä lähtien vahvin shakkiohjelma. NNUE:ta hyödyntävä Stockfish-versio osoittautui paremmaksi kuin aiempi käsintehtyä arviointifunktiota käyttävä versio, vaikka NNUE:n käyttämä laskenta-aika lähes puolitti haun nopeuden [4].

NNUE on LC0:n ja AlphaZeron hyödyntämiin verkkoihin verrattuna pieni verkko. Se koostuu vain kolmesta piilotetusta kerroksesta. Syötekoodaus on jaettu kahteen osaan. Ensimmäisessä osassa on kaikki nappulat suhteessa valkoisen kuninkaaseen ja toisessa osassa on kaikki nappulat suhteessa mustan kuninkaaseen. Molemmissa osissa on tieto siitä, kummalla pelaajalla siirtovuoro on. Pelaajilla on 5 uniikkia nappulatyyppiä kuninkaan lisäksi. Tällöin syötteiden määrä on shakissa

$K_{sq} \cdot P_{sq} \cdot P_n \cdot C$, jossa K_{sq} on kuninkaan mahdolliset ruudut, P_{sq} tietyn tyyppisen pelinappulan mahdolliset ruudut, P_n pelinappulatyyppien määrä ja C värien määrä. Shakissa NNUE saa siis syötteitä 40960. NNUE:ssa on lisäksi shōgia varten 64 syötettä, jotka ovat shakissa aina 0, joten ne voidaan jättää huomioimatta. [4] Syötteet ovat bittejä ja ne saavat arvon seuraavasti:

(valkoinen kuningas ruudulla A1, valkoinen sotilas, A1)

(valkoinen kuningas ruudulla A1, valkoinen sotilas, A2)

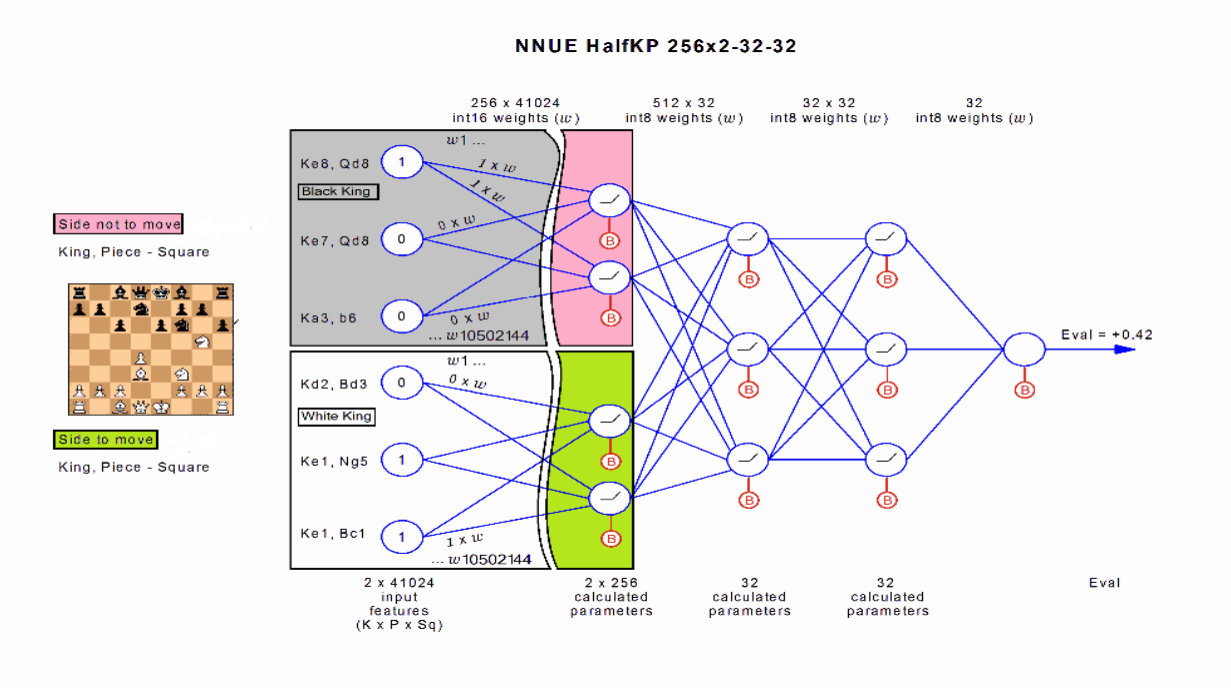
...

(valkoinen kuningas ruudulla A2, valkoinen sotilas A1)

Eli yleisesti:

$$(K_{64},\ P,\ P_{64})$$

Syöte saa arvon 1, jos kuningas sijaitsee ruudulla K_{64} ja tietyn tyyppinen pelinappula P sijaitsee ruudulla P_{64} ja muuten arvon 0. Koska kuningas voi olla vain yhdellä ruudulla, suurin osa syötteistä saa arvon 0. Yksi siirto aiheuttaa siis kahden syötebitin muutoksen, jos siirto ei ollut kuninkaan siirto, minkä lisäksi kuningas siirtyy harvoin shakissa. HalfKp voi käyttää samoja painoja molemmille puolille symmetrian ansiosta. Esimerkiksi (valkoinen kuningas A1:ssä, valkoinen sotilas B2:ssa) -paino on sama kuin (musta kuningas H1:ssä, musta sotilas H2:ssa) -paino. Näiden ominaisuuksien seurauksena tyypillinen siirto shakissa aiheuttaa suhteellisen pienen muutoksen neuroverkon syötteeseen. [3] Lopulta neuroverkko antaa peliasemalle yhden skaalariarvon. Neuroverkko voidaan kouluttaa ohjatulla oppimisella (engl. supervised learning), missä verkolle annetaan opetusaineistoksi valmiiksi arvioituja peliasemoita. Stockfish on käyttänyt NNUE:n oppiaineistona LC0:n pelaamia ja arvioimia koulutuspelejä. [17]



Kuva 5 NNUE:n rakenne [18]

Syötekoodausta kutsutaan nimellä HalfKp (Half-King-Piece relationship) ja se keskittyy nappuloiden ja kuninkaiden väliseen suhteeseen, minkä Klein spekuloi olevan jäänne shōgista. Shōgin luonteessa on kaksi asiaa, jotka voisivat selittää tämän: kuninkaan turvallisuuteen tarvitsee panostaa huomattavasti enemmän resursseja ja pelinappulat kykenevät liikkumaan keskimäärin vähemmän ruutuja siirrossa kuin shakissa, minkä seurauksena pelinappuloiden etäisyys kuninkaaseen on huomattavasti merkityksellisempää.

Empiirisesti on kuitenkin havaittu, että syötekoodaus toimii myös shakissa. NNUE:n nopeuden ansioista sitä voidaan hyödyntää alpha-beta-haussa. Mitä vähemmän aikaa neuroverkko käyttää, sitä enemmän aikaa haulle jää. Toisin kuin LC0:n neuroverkko, NNUE:n rakenne on hyvin optimoitu CPU:lle, minkä ansiosta sitä voidaan käyttää kotitietokoneilla tai jopa älypuhelimella. [4]

3.6 Monte carlo –puuhaku

Eloluvultaan parhaimmat shakkiohjelmat ovat kaikki perustuneet alpha-beta-hakuun. Alpha-beta-hakua on myös sovellettu menestyneesti monessa pelissä kuten shōgissa, tammessa ja othellossa. Parhaimpien ihmispelaajien tasolla olevaa alpha-beta-hakuun perustuvaa go-ohjelmaa ei ole kuitenkaan onnistuttu luomaan. Gon keskimääräinen haarautumiskerroin on 250, joka on huomattavasti suurempi kuin shakin. Myös arviointifunktion kehittäminen on vaikeaa. Shakissa materiaalietua voidaan käyttää perustana arviointifunktiolle, mille ei ole vastinetta gossa. Tähän ongelmaan yksi ratkaisu on haku, johon ei tarvita arviointifunktiota. [19]

Monte Carlo -simulaatioihin perustuvia algoritmeja on esitelty jo 1950-luvulta lähtien. Näissä simulaatioissa pyritään luomaan stokastinen prosessi, joka sisältää satunnaismuuttujan, jonka odotusarvo toimii ongelman ratkaisuna [20]. Rémi Coulom esitti vuonna 2006 tutkimusartikkelissaan haun, jota kutsutaan Monte Carlo -puuhauksi [18]. Monte Carlo –puuhaku jaetaan neljään vaiheeseen: valintaan (engl. selection), laajennukseen (engl. expansion), simulaatioon ja taaksepäin välitykseen (engl. backpropagation).

Valinnassa aloitetaan nykyisestä peliasemasta eli juurisolmusta R , jonka jälkeen valitaan peräkkäisiä lapsisolmuja, kunnes saavutetaan lehtisolmu L . Lehtisolmu on mikä tahansa mahdollinen siirto, josta ei ole vielä suoritettu simulaatioita. Tässä kohtaa voidaan vaikuttaa lapsisolmujen valintaan siten, että pelipuu laajenee kohti lupaavimpia siirtoja.

Jos valinnan lehtisolmu L ei ole päätössolmu, tehdään laajennus muodostamalla yksi tai useampi lapsisolmu, joista valitaan jokin solmu C . Lapsisolmut ovat mahdollisia siirtoja, jota voidaan tehdä solmussa L

Simulaatiossa suoritetaan tilastollinen laskelma peleistä, jotka alkavat solmusta C . Pelejä pelataan mielivaltainen määrä päätösolmuun saakka satunnaisilla siirroilla. Vaikka

simulaatioita tarvitsee tehdä paljon, laskennan määrä on silti vähäistä, koska se vaatii vain siirtojen muodostuksen.

Viimeisessä vaiheessa välitetään (engl. Backpropagation) tilastollinen tieto solmusta C takaisin pelipuuta pitkin juurisolmulle. Tilastollinen tieto on simulaatiossa pelattujen pelien voittoprosentti. Laajennukset voidaan tehdä siten, että korkeimman voittoprosentin omaavat solmut laajennetaan ensin. [4]

Alpha-beta-hakuun verrattuna Monte Carlo -puuhaku käyttää vähemmän pelikohtaista tietoa (engl. domain knowledge) [4]. Koska shakissa on monta erilaista pelinappulaa ja erikoissääntöä, siirronmuodostus on laskennallisesti vaativampaa kuin esimerkiksi gossa, jossa Monte Carlo -puuhaku on osoittautunut paremmaksi hakutavaksi. Tämän lisäksi shakissa on historiallisesti aina ollut kohtulliseen hyviä arviointifunktiota ja pelikohtaiseen tietoon perustuvia heuristisia menetelmiä karsia pelipuuta, joten alpha-beta-haku on säilynyt hallitsevana hakumenetelmänä.

4 AlphaZero ja LeelaChessZero

AlphaGo voitti gon maailmanmestarin Lee Sedolin vuonna 2016. Tätä ottelua voidaan pitää gon vastineena Deep Bluen ja Garry Kasparovin välisestä ottelusta. AlphaGo oli rakenteeltaan hyvin erilainen ohjelma kuin sen vastineet shakissa ja shōgissa. AlphaGo koostui viidestä eri neuroverkosta, jonka lisäksi se käytti Monte Carlo -puuhakua etsimään siirtoja [4]. Sitä suoritettiin yhteensä 1920 CPU:lla ja 280 GPU:lla [21].

AlphaGota aluksi koulutettiin käyttäen ohjattua oppimista (engl. Supervised learning), jonka opetusaineistona käytettiin parhaiden ihmispelaajien pelaamia pelejä. Tämän jälkeen AlphaGota koulutettiin pelaamalla pelejä itseään vastaan käyttäen vahvistusoppimista (engl. reinforcement learning). AlphaGon kehittäjä DeepMind jalosti AlphaGota eteenpäin siten, että ohjelmaan jäi vain yksi neuroverkko, jota koulutettiin itseoppimaan hyödyntäen vahvistusoppimista, mistä syntyi AlphaGoZero. DeepMind huomasi, että vaikka AlphaGo oppi nopeammin ennustamaan opetusaineistonsa lopputuloksia, AlphaGoZero oli pidemmän harjoitusjakson jälkeen parempi ennustamaan lopputuloksia saman opetusaineiston peleissä. AlphaGon eloluku jäi noin 3500 tasolle, kun taas AlphaGoZeroa saatiin koulutettua noin 4000 eloluvun tasolle. [4] Tämä kertoo neuroverkkojen kyvystä oppia ominaisuuksia tai piirteitä gosta, joita edes parhaimmat ihmispelaajat eivät kykene näkemään.

DeepMind esitteli AlphaGoZeroon pohjautuvan AlphaZeron, joka kykenee pelaamaan shakkia ja shōgia [22]. AlphaZero sisältää hyvin vähän pelikohtaista tietoa: pelin säännöt ja pelin tyypillinen pituus. Se koostuu yhdestä neuroverkosta, joka saa syötteeksi peliaseman. Peliaseman jokaiselle siirrolle annetaan todennäköisyys, joka kuvaa sen todennäköisyyttä olla paras siirto peliasemassa, ja peliasemalle annetaan ennustus pelin lopputuloksesta.

AlphaZero pelaa hakemalla muunnellulla Monte Carlo –puuhaulla parhaan siirron. Neuroverkko ohjaa haun seuraavasti: valintavaiheessa neuroverkolta pyydetään siirtojen todennäköisyydet. Näistä valitaan siirrot, joilla on parhaat todennäköisyydet. Koska simulaation tarkoitus on kerätä tilastollinen ennuste siirron voittoprosentille, se voidaan myös korvata. Simulaatiovaihe korvataan kokonaan pyytämällä lapsisolmujen voittoennuste neuroverkolta. [3]

AlphaGoZeroa koulutettiin peluuttamalla sitä itseään vastaan ja sen häviöfunktiona (engl. Loss function) käytettiin $(z - v)^2 - \pi^T \ln(p) + c\left|\left|\theta\right|\right|^2$, missä

z : Pelin oikea lopputulos

v : Neuroverkon antama ennuste lopputulokselle

π^T : Neuroverkon antamat siirtojen todennäköisyydet haun jälkeen

p : Neuroverkon antamat siirtojen todennäköisyydet ennen hakua

$c||\theta||^2$: Regularisaatiotermi (engl. regularization term)

Koulutuksessa optimoidaan siis kahta asiaa: neuroverkon kykyä ennustaa pelin lopputulosta peliasemasta ja siirtojen todennäköisyyksien eroa ennen ja jälkeen haun. [21] Mitä paremmin neuroverkko arvioi pelitilanteita, sitä paremmin se ohjaa hakua.

AlphaZero ei kilpaillut shakissa Stockfishin tavoin kilpailuissa, joten sen taitotasoa on vaikea arvioida. DeepMind kertoi peluuttaneensa sitä Stockfish 8 -versiota vastaan 100 ottelun sarjassa, joista Alphazero voitti 28 ja loput 72 peliä oli tasapelejä [21]. Tämä viittaa siihen, että Alphazero olisi tuolloin ollut noin 100 elopistettä vahvempi kuin Stockfish, mikä olisi ollut mullistava lopputulos, sillä Stockfishin käyttämää alpha-beta hakua on optimoitu yli 60 vuoden ajan.

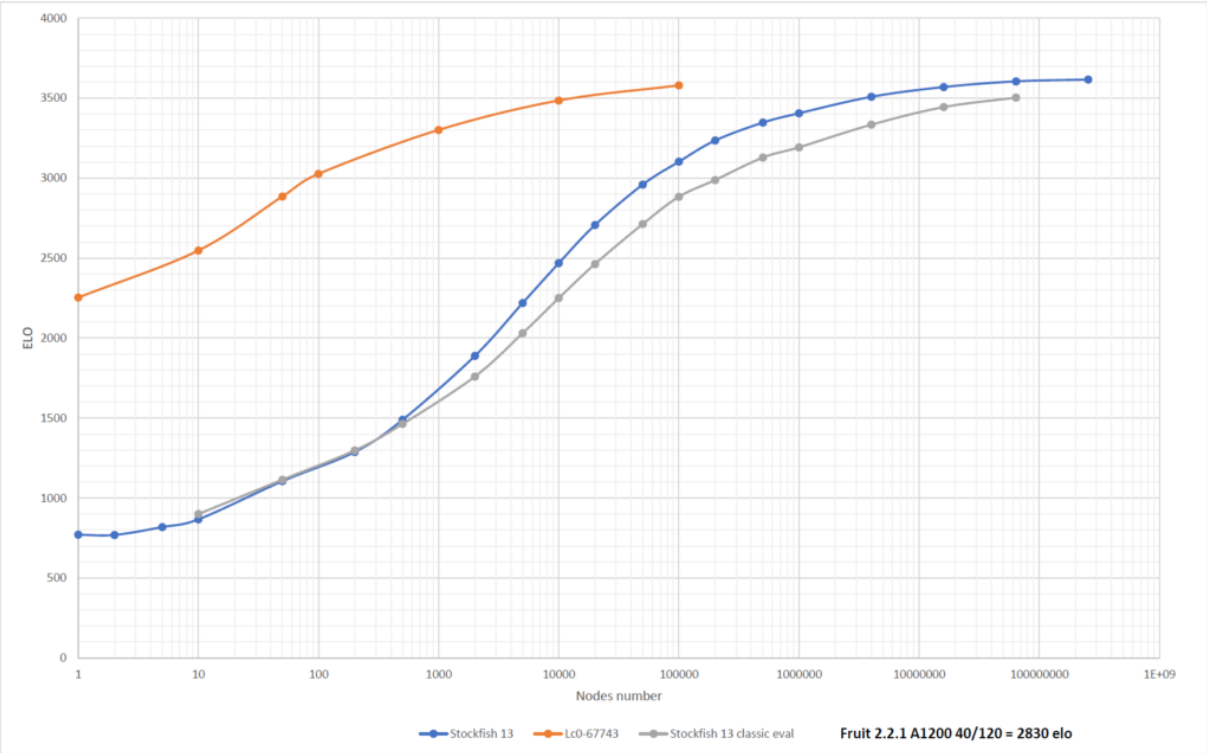
DeepMindin asettama vertailu oli kuitenkin kyseenalaista. Sen sijaan, että ajankäytön hallinta olisi ollut shakkiohjelmalla, kummallekin annettiin minuutti aikaa tehdä jokainen vuoro. Tämä ei ole tyypillistä shakkiturnauksissa, vaan yleisesti pelaajille annetaan tietty määrä aikaa koko pelille, jonka pelaajat saavat käyttää miten haluavat. Lisäksi Stockfishiin kuuluvat aloitus- ja loppupelikirjat oli poistettu käytöstä ja sen transpositiotaulun koko oli rajoitettu yhteen gigatavuun. Näille rajoituksille DeepMind ei antanut perusteita. Merkittävää kuitenkin oli AlphaZeron kyky ymmärtää peliasemia. Sen haku tutkii noin 100 kertaa vähemmän peliasemia kuin Stockfish ja ilman hakua AlphaZeron neuroverkko kykenee pelaamaan noin 2400-2500 eloluvun tasolla. [4]

AlphaZeron käytännön toteutusta tai sen lähdekoodia ei julkistettu, joten DeepMindin esittämien konseptien ja ideoiden pohjalta luotiin avoimen lähdekoodin projekti LeelaChessZero vuonna 2018 [23]. Se ehti hetkellisesti olla paras shakkiohjelma 2019-2020, ennen NNUE:n lisäystä Stockfishiin.

Leelan konvoluutioneuroverkko (engl. convolutional neural network) korvattiin vuonna 2022 transformer-arkkitehtuurilla. Konvoluutioverkon heikkouksiin kuului sen puutteellinen ymmärrys pitkien etäisyyksien riippuvaisuuksista, koska käytännössä jokainen ruutu analysoi iteratiivisesti naapuriruutujen informaatiota ja käyttää tätä informaatiota oman peliesityksen

tarkentamiseen. Esimerkiksi jotta a1-ruutu voisi "oppia" h8-ruudun nappulasta, informaation on kuljettava ainakin 7 ruudun matka edestakaisin. Transformeri-arkkitehtuurin ansioista voidaan määritellä etäisyys uudelleen, joka ei ole enää pelkkä euklidinen etäisyys, vaan etäisyys riippuu myös shakkinappulan tyypistä. Esimerkiksi etäisyys (A1, A8) on sotilaalle 5-6 siirtoa, kun taas tornille mahdollisesti vain 1. Lisäksi shakkiruuduille voidaan antaa erilainen merkitys riippuen siitä, että onko peli avoin vai suljettu. Transformeriin pohjautuvan verkon eloluku on noin 270 pistettä suurempi ilman hakua, kuin konvoluutioneuroverkon ilman hakua. [24]

5 Pohdintaa



Kuva 6 [25] Shakkiohjelman eloluvun suhde sen käsittelemien solmujen määrään

Shakkiohjelmien kyky tutkia tietty määrä solmuja määrä kasvaa, kun lisätään laskentatehoa. Tutkittujen solmujen määrä kasvattaa LC0:n, Stockfishin (käsintehty arviointifunktio) ja Stockfishin (NNUE) eloa kuvassa 6 esitettyjen kuvaajien mukaisesti. Koska käyrät tasottuvat, voidaan tehdä johtopäätös, että parempien shakkiohjelmien aikaansaamiseksi ei riitä loputtomasti se, että lisätään laskentatehoa, vaan tarvitaan uusia menetelmiä ja kehityksiä myös ohjelmassa. [25]

Kuvassa 6 esitetty kuvaaja kertoo myös neuroverkkojen shakkiymmärryksestä. Ilman hakua Stockfish saa NNUE:n antamalla arvioilla huomattavasti heikomman elotason kuin Leela saa ilman sen hakua. Sen laskennallinen keveys kuitenkin mahdollistaa hyvin syvän ja laajan haun, mikä ei ole mahdollista Leelalle [4]. Lisäksi, koska Leelan haku tehdään CPU:lla ja neuroverkon laskenta GPU:lla tai vastaavalla laitteistokiihdyttimellä, suorittimien välisestä kommunikaatiosta voi muodostua merkittävä pullonkaula. Stockfish voidaan suorittaa pelkällä suorittimella, joten tätä ongelmaa ei ole [4]. Stockfish on ollut ylivoimainen etenkin TCEC kilpailuissa viimeiset 10 vuotta ja NNUE lisäsi ylivoimaisuutta entisestään. Muita merkittäviä nykyaikaisia shakkiohjelmia ovat esimerkiksi Torch, KomodoDragon, Berserk ja Obsidian, jotka ovat kaikki alpha-beta hakijoita.

Sekä Stockfishin NNUE:n että Leelan ideat ovat alun perin muista lautapeleistä, mikä osoittaa, kuinka hyödyllistä on yhdistää eri lautapelien tutkimusta. Jatkotutkimuksen kohteena voisi olla NNUE:n sekä Leelan täysi “shakillistaminen”. Monte Carlo -puuhaun käyttö muuallakin kuin Leelassa voisi olla hyödyllistä, sillä NNUE on osoittanut, että ihmisen pelikohtainen tieto shakista saattaa monella tapaa olla puuttellinen.

6 Yhteenveto

Tutkielman ensimmäinen tutkimuskysymyksen tavoitteena oli selvittää shakkiohjelmien toteutuksia, niiden käyttämiä algoritmeja ja menetelmiä. Tutkimusta ja shakkiohjelmia löytyi runsaasti 1950-luvulta alkaen. Suurimman osan tietokoneshakin historiasta shakkiohjelmat on toteutettu alpha-beta-karsintaan pohjautuvalla haulla. Tutkimus on pitkälti keskittynyt erilaisiin optimointimenetelmiin. Näistä merkittävimpiin kuuluu heuristinen siirtojen järjestely, iteratiivinen syventäminen, transpositiotaulu, nollasiirtokarsinta ja päävariaatiohaku. Tutkielman toisessa tutkimuskysymyksessä pohdittiin neuroverkkojen hyödyntämistä shakkiohjelmissa. Neuroverkkoja yleisesti käytetään arvioimaan pelitilanteita ja nykyiset shakkiohjelmat voidaan jakaa kahteen kategoriaan. Shakkiohjelmat, jotka käyttävät paljon resursseja syvään hakuun käyttäen laskennallisesti kevyttä neuroverkkoa. Toinen kategoria on shakkiohjelmat, jotka käyttävät laskennallisesti raskasta neuroverkkoa ja matalaa hakua.

Alpha-beta-hakuun on keksitty lukuisia karsintamenetelmiä, joiden on todettu lisäävän shakkiohjelman elolukua. Jotkut näistä eivät ole yleistettäviä kuten alpha-beta-karsinta, vaan ne parantavat suorituskykyä vain tiettyjen heurististen menetelmien yhdistelminä. Monien kehityksien tuomia hyötyjä on siis vaikea tutkia.

Tutkielman lopullisiksi lähteiksi tuli valituiksi sellaiset lähteet, joissa tutkitut ja esitellyt menetelmät ovat edelleen käytössä nykypäivänä. Ensimmäisiä shakkiohjelmia kehitettiin jo tietokoneiden kehityksen alkuvaiheessa, joten shakkiohjelmien toteutuksissa oli huomioitava esimerkiksi muistin rajallisuus. Stockfishin transpositiotaulun maksimikoko on 32 teratavua, mikä on todennäköisesti enemmän kuin kaikkien 1960-luvun tietokoneiden yhteenlaskettu muisti. Laitteistoon keskittyvät lähteet myös karsittiin, sillä tutkimuksen painopiste on ohjelmistossa. Huomattava määrä tutkimusta on myös tehty käsintehtyjen arviointifunktioiden luomiseen. Neuroverkot ovat kuitenkin osoittautuneet olevan merkittävästi parempia arviointifunktiona, joten käsintehdyt arviointifunktiot jäivät tutkimuksen ulkopuolelle.

Deep Bluen voiton myötä kiinnostus shakkiohjelmien tutkimukseen saattoi vähetä, kun tekoälytutkijat siirtyivät ratkaisemattomampien haasteiden, kuten gon, pariin. Gota pidettiin tekoälyn avoimena ongelmana aina vuoteen 2016 asti, jolloin AlphaGo voitti maailmanmestari Lee Sedolin. Sekä gossa että shakissa on kuitenkin edelleen paljon tutkittavaa. Vaikka ohjelmat ovat kehittyneet paremmiksi kuin ihmispelaajat, shakkia ja gota ei ole vielä ratkaistu täydellisesti. Nykyisin shakin merkitys tekoälytutkimuksessa on vähentynyt, mutta shakkiohjelmien kehitys kiinnostaa edelleen shakkiharrastajia ja -tutkijoita.

Lisäksi shakkiohjelmista voi löytyä hyödyllisiä optimointimenetelmiä ja algoritmeja muihin ongelmiin, erityisesti sellaisiin, joissa voidaan soveltaa minimaxia tai muita haku- ja arviointialgoritmeja.

Lähteet

[1] H. Davidson, *A Short History of Chess*. Greenberg, eISBN 978-0-307-82829-3.

[2] C. Shannon, "*XXII. Programming a computer for playing chess*," London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science, vsk. 41, nro 314, ss. 256–275, marraskuu 1950. doi: 10.1080/14786445008521796.

[3] J. Kister, P. Stein, S. Ulam, W. Walden ja M. Wells, "*Experiments in Chess*," Journal of the ACM, vsk. 4, nro 2, ss. 174–177, huhtikuu 1957. https://doi.org/10.1145/320868.320877.

[4] D. Klein, "*Neural networks for chess*," arXiv, syyskuu 2022. Saatavilla: https://doi.org/10.48550/arXiv.2209.01506. Viitattu: marraskuu 2024

[5] F.-h. Hsu, *Behind Deep Blue*: *Building the Computer that Defeated the World Chess Champion*, uudistettu painos. Princeton, NJ: Princeton University Press, 2002.

[6] A. Bozhinovski ja F. Jankuloski, "*Chess as played by artificial intelligence*," Proc. ICT Innovations 2020, syyskuu 2020. Saatavilla: http://hdl.handle.net/20.500.12188/9487.

[7] H. Berliner, "*The B* Tree Search Algorithm: A Best-First Proof Procedure*," Readings in Artificial Intelligence, 1981, ss. 79–87. Saatavilla: https://doi.org/10.1016/B978-0-934613-03-3.50010-6.

[8] G. C. Stockman, "*A minimax algorithm better than alpha-beta?*," Artificial Intelligence, vsk. 12, nro 2, ss. 179–196, elokuu 1979. Saatavilla: https://doi.org/10.1016/0004-3702(79)90016-X.

[9] N. Thompson, G. Manso ja S. Ge, "*The importance of (exponentially more) computing power*," Academy of Management Annual Meeting Proceedings, heinäkuu 2023. Saatavilla: https://doi.org/10.5465/AMPROC.2023.365bp.

[10] Stockfish-kehittäjät, "*Stockfish*." Saatavilla: https://github.com/official-stockfish/. Viitattu: marraskuu 2024.

[11] D. Beal, "*A generalised quiescence search algorithm*," Artificial Intelligence, vsk. 43, nro 1, ss. 85–98, huhtikuu 1990. doi: 10.1016/0004-3702(90)90072-8.

[12] J. Schaeffer, "*The history heuristic and alpha-beta search enhancements in practice*," IEEE Trans. Pattern Anal. Mach. Intell., vsk. 11, nro 11, ss. 1203–1212, marraskuu 1989. doi: 10.1109/34.42858.

[13] H. Iida, M. Sakuta ja J. Rollason, "*Computer shogi*," Artificial Intelligence, vsk. 134, nro 1–2, ss. 121–144, tammikuu 2002. doi: 10.1016/S0004-3702(01)00157-6.

[14] T. Marsland, "*The anatomy of chess programs*," AAAI Spring Symp. Series, Stanford, CA, 1997. Saatavilla: https://cdn.aaai.org/Workshops/1997/WS-97-04/WS97-04-006.pdf.

[15] D. M. Breuker ja J. W. H. M. Uiterwijk, "*Transposition tables in computer chess*," Department of Computer Science, University of Limburg

[16] M. S. Campbell ja T. A. Marsland, "*A comparison of minimax tree search algorithms*," Artificial Intelligence, vsk. 20, nro 4, ss. 347–367, 1983. doi: 10.1016/0004-3702(83)90001-2.

[17] "*NNUE training data*.". Saatavilla: https://robotmoon.com/nnue-training-data/. Viitattu: marraskuu 2024.

[18] "*Stockfish NNUE layers*." Saatavilla: https://www.chessprogramming.org/File:StockfishNNUELayers.png. Viitattu: marraskuu 2024.

[19] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling ja P. Rohlfshagen, "*A survey of Monte Carlo tree search methods*," IEEE Trans. Comput. Intell. AI Games, vsk. 4, nro 1, ss. 1–43, maaliskuu 2012. doi: 10.1109/TCIAIG.2012.2186816.

[20] R. Coulom, "*Efficient selectivity and backup operators in Monte-Carlo tree search*," Computers and Games, L. K. T, vsk. 4630, ss. 72–83, 2006. doi: 10.1007/11780636_7.

[21] "*Showdown*," The Economist, 12. maaliskuuta 2016. Saatavilla: https://www.economist.com/science-and-technology/2016/03/12/showdown. Viitattu: marraskuu 2024.

[22] D. Silver ym., "*Mastering chess and shogi by self-play with a general reinforcement learning algorithm*," DeepMind. Saatavilla: https://doi.org/10.48550/arXiv.1712.01815.
Viitattu: marraskuu 2024

[23] LeelaChessZero-kehittäjät, "*LC0*." Saatavilla: https://github.com/LeelaChessZero/lc0. Viitattu: marraskuu 2024.

[24] "*Transformer progress*." Saatavilla: https://lczero.org/blog/2024/02/transformer-progress/. Viitattu: marraskuu 2024.

[25] "*Stockfish and Lc0*," test at different number of nodes." Saatavilla: https://www.melonimarco.it/en/2021/03/08/stockfish-and-lc0-test-at-different-number-of-nodes/. Viitattu: marraskuu 2024.