Tampere University

Tuomo Sillanpää

MONTE CARLO -PUUHAKUA KÄYTTÄVIEN TEKOÄLYMENETELMIEN SOVELTUVUUS VUOROPOHJAISIIN STRATEGIAPELEIHIN

Informaatioteknologian ja viestinnän tiedekunta

Pro gradu -tutkielma

Kesäkuu 2023

Tiivistelmä

Tuomo Sillanpää: Monte Carlo -puuhakua käyttävien tekoälymenetelmien soveltuvuus vuoropohjaisiin strategiapeleihin
Pro gradu -tutkielma
Tampereen Yliopisto
Master's Programme in Software Development
Kesäkuu 2023

Tässä tutkielmassa tarkastellaan Monte Carlo -puuhaun soveltuvuutta vuoropohjaisten strategiapelien tekoälyratkaisuihin kirjallisuuskatsausta hyödyntäen. Aluksi esitellään sekä minimax-algoritmi että Monte Carlo -puuhaku suosittuine muunnelmineen, ja sen jälkeen perehdytään tarkemmin neljään vuoropohjaiseen strategiapeliin: Shakkiin, go-lautapeliin, pokeriin ja Magic: The Gathering -keräilykorttipeliin.

Kunkin neljän pelin kohdalla tutustutaan kyseisen pelin tekoälylle asettamiin haasteisiin, olemassa oleviin tekoälyratkaisuihin ja etenkin Monte Carlo -puuhakua hyödyntäviin tekoälytoimijoihin. Lopuksi luodaan vielä lyhyt katsaus joukkoon Monte Carlo -puuhakua hyödyntäviä tekoälyratkaisuja muiden vuoropohjaisten strategiapelien kohdalla.

Huomataan, että Monte Carlo -puuhakua käyttämällä saavutetaan sen yleispätevän luonteen vuoksi usein merkittäviä hyötyjä etenkin sellaisissa peleissä, joille mielekkään evaluaatiofunktion kirjoittaminen on hankalaa.

Avainsanat: tekoäly, AI, artificial intelligence, algoritmit, algorithms, Monte Carlo -puuhaku, Monte Carlo Tree Search, MCTS, vuoropohjaiset strategiapelit, turn-based strategy games

Sisällysluettelo

1 Johdanto

Puu on tietorakennetyyppi, joka koostuu hierarkkisesti toisiinsa kiinnitetyistä solmuista, jotka sisältävät jotakin informaatiota. Vuoropohjaisten pelien tekoälytoteutuksissa puurakenteita käytetään yleensä haarautuvan suorituksen logiikan toteuttamiseen. Tässä tutkielmassa tutustutaan puuhakualgoritmeihin strategiapelien tekoälytoimijoiden ratkaisuissa, kartoittaen erityisesti viime vuosikymmeninä laajalti sovelletun Monte Carlo -puuhaun mahdollisuuksia kirjallisuuskatsaukseen pohjaten.

Luvussa 2 tarkastellaan vuoropohjaisissa strategiapeleissä käytettyjen tekoälytoteutusten yleisimpiä puuhakualgoritmeja. Minimax-algoritmin eri muunnelmien voidaan katsoa muodostavan algoritmisen pohjan etenkin kahden pelaajan vuoropohjaisten strategiapelien tekoälytoimijoille. Vakiomuotoisen minimaxin rajat tulevat kuitenkin nopeasti vastaan peleissä, joissa korkea *haarautumiskerroin* (branching factor) rajoittaa laskentateho- ja muistikapasiteettisyistä puurakenteen syvyyttä. Haarautumiskertoimella tarkoitetaan tässä puurakenteen jokaisen solmun keskimääräistä lapsisolmujen määrää. Koska koko puurakennetta ei useimpien ei-triviaalien pelien tapauksessa käytännössä voida tutkia, tulee lehtisolmujen pelitilan arvo tekoälyn tavoitteiden kannalta arvioida evaluaatiofunktiolla. Tällaisen evaluaatiofunktion laatiminen vaatii pelikohtaisen asiantuntijatietämyksen formalisointia ohjelmakoodimuotoon, mikä on monien pelien kohdalla erittäin hankalaa, ellei lähes mahdotonta, toteuttaa tyydyttävästi.

Monte Carlo -puuhakualgoritmi (Monte Carlo Tree Search, MCTS) puolestaan suorittaa pelitilapuussa satunnaisia simulaatioita, jolloin peliä pelataan toistuvasti loppuun asti satunnaisia toimintoja suorittamalla. Jokaisen läpipeluun arvo, joka määräytyy pelin tavalla tai toisella päättävän terminaalisolmun arvosta, propagoidaan tämän jälkeen siihen johtaneen polun jokaiseen solmuun. Menetelmällä ei siis käydä läpi pelitilapuuta kokonaisuudessaan, vaan siitä kerätään tällaisen "simulaationäytteenoton" avulla tilastollista dataa, jota voidaan hyödyntää tekoälyn päätöksenteossa. Koska jokainen simulaatio suoritetaan terminaalisolmuun asti, ei evaluaatiofunktiolle ole tarvetta. Tämä tekee Monte Carlo -puuhausta yleispätevän algoritmin, sillä tekoälyn toteutuksen täytyy tuntea ainoastaan pelin säännöt, jotta se kykenee valitsemaan laillisia toimintoja. Kattavampi aluekohtainen asiantuntijatietämys pelistä ei ole tarpeen.

Tutkielman tutkimusmenetelmänä on käytetty kirjallisuuskatsausta, jonka rakennetta on lyhyesti kuvattu luvussa 3. Relevanttia kirjallisuutta tekoälyratkaisuiden ja MCTS:n käytöstä vuoropohjaisissa strategiapeleissä haettiin useasta eri tietokannasta joukolla hakulausekkeita. Aineistoksi pyrittiin valikoimaan shakin, go-pelin, pokerin ja Magic: the Gathering -keräilykorttipelin osalta noin viisi mielenkiintoisinta ja laadukkainta lähdettä. Lisäksi valikoitiin joukko lähteitä, joissa käsiteltiin joitakin muita vuoropohjaisten pelien MCTS-ratkaisuja.

Luvussa 4 tarkastellut shakki ja go ovat molemmat vanhoja ja maineikkaita lautapelejä, joiden tekoälyä on tutkittu ja kehitetty jo pitkään. Nämä kaksi peliä ovat tekoälykontekstissa pintapuolisesti hyvin samankaltaisia - kummassakin kaksi pelaajaa pelaa vastakkain pelilaudalla, pelaajilla on jatkuvasti täydellinen tieto pelin tilasta, ja peli ei sisällä mitään stokastisia eli satunnaisia elementtejä. Go on kuitenkin osoittautunut huomattavasti shakkia haasteellisemmaksi tekoälyratkaisuiden osalta. Kaksi muuta lukuun 4 valittua tarkasteltavaa peliä, pokeri ja Magic: The Gathering (MTG), taas poikkeavat shakista ja go:sta huomattavasti, sillä peleissä esiintyy sekä epätäydellistä informaatiota että stokastisuutta. Lisäksi pokerin tapauksessa pelaajia on useampi kuin kaksi, ja MTG:ssä pelin kortit voivat vapaasti muokata pelin sääntöjä. Tämä tekijät aiheuttavat merkittäviä haasteita tekoälyratkaisuille, joihin tutustutaan luvussa kolme jokaisen neljän pelin kohdalla. Lisäksi kunkin pelin tapauksessa käydään kirjallisuuteen perustuen läpi MCTS-algoritmin sovelluksia pelin tekoälyratkaisuihin.

Luvussa 5 tarkastellaan suppeammin MTCS-pohjaisia tekoälyratkaisuja Carcassonnen, Diplomacyn, Settlers of Catanin sekä Spades- ja Hearthstone-korttipelien osalta. Lopuksi luku 6 päättää tutkielman tehtyjen havaintojen pohdinnalla.

2. Puuhaku-algoritmeja

Tässä tutkielmassa keskitytään ensisijaisesti puuhakualgoritmeihin vuoropohjaisissa strategiapeleissä, ja tarkemmasta tarkastelusta rajataan pois muut menetelmät, kuten neuroverkot ja koneoppiminen. Erityisesti tutustutaan Monte Carlo -puuhakualgoritmiin ja sen käyttökohteisiin sekä etuihin muihin tekniikoihin verrattuna tai yhdistettynä.

Pelitekoälyn kontekstissa puuhakualgoritmien toimintaa voidaan kuvata yleisesti seuraavasti: Puurakennetta lähdetään rakentamaan jostakin pelin tilasta, yleensä aloituspelitilasta. Tässä tilassa jokainen aktiivisen (pelivuorossa olevan) pelaajan suorittama mahdollinen toiminto luo puuhun uuden solmun, eli uuden pelitilan, ja niin edelleen, kunnes teoriassa puurakenne sisältää kaikki pelin mahdolliset tilat sen alusta loppuun asti. Tällöin valmiista puusta voidaan nähdä, mitkä siirrot ovat missäkin tilassa pelaajalle optimaalisia. Tällainen täydellisen puun rakentaminen on kuitenkin käytännössä mahdollista vain hyvin yksinkertaisissa peleissä, joissa on hyvin matala *haarautumiskerroin*, kuten esimerkiksi 3x3-ruudukon ristinollassa. Useimmissa monimutkaisemmissa peleissä mikään saatavilla oleva laskentateho ja/tai muisti ei mahdollista tällaista lähestymistapaa.

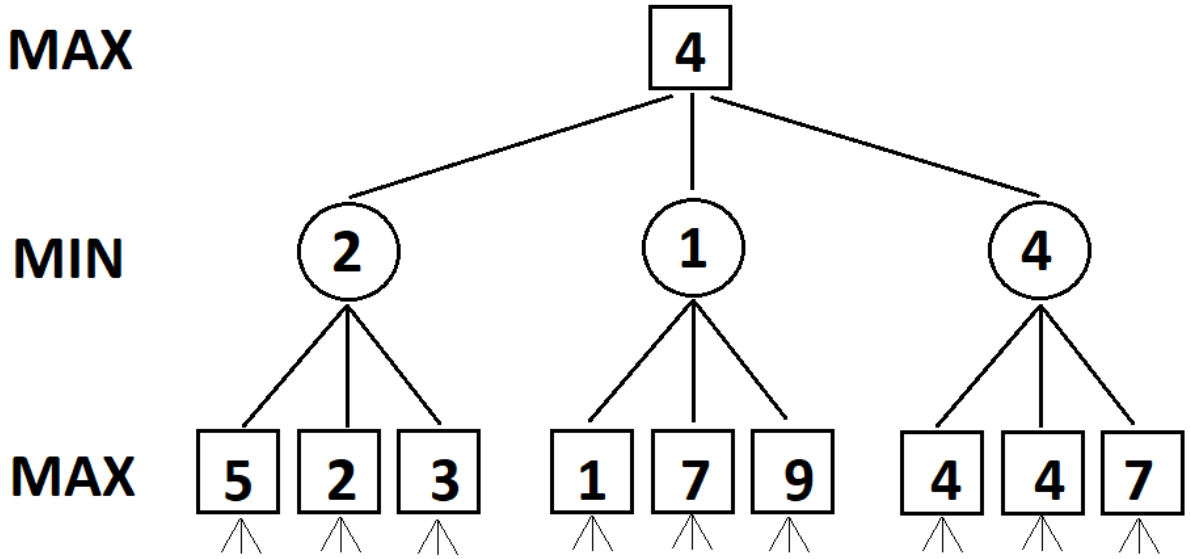
Puuta rakennettaessa voidaan edetä joko syvyys- tai leveyssuunnassa. Syvyyssuunnassa etenevän puun laajennuksen englanninkielinen nimitys on *Depth First Search*, lyhennettynä DFS. DFS-haussa jokaisesta solmusta laajennetaan vain ensimmäinen lapsisolmu, kunnes saavutaan *terminaalisolmuun* eli tilaan, jossa peli päättyy. Tämän jälkeen, mikäli tarpeen, palataan puussa takaisin kunnes löytyy uusi lapsisolmu laajennettavaksi. Leveyssuunnassa etenevä haku puolestaan on nimeltään *Breadth First Search*, eli BFS. Tällöin kunkin tason kaikki solmut käydään läpi ennen seuraavalle solmutasolle siirtymistä.

Aiemmin mainituista käytännön laskentateho- ja muistirajoituksista johtuen tällaiset haut eivät kuitenkaan sellaisenaan ole kelvollisia ratkaisuja useimmissa tekoälyratkaisuissa. Tällöin hakua voidaan rajata esim. syvyyssuunnassa siten, että puuta ei laajennetakaan terminaalisolmuihin, vaan ainoastaan johonkin määritettyyn syvyyteen asti, jolloin puhutaan *syvyysrajoitetusta* hausta. Tällöin

lehtisolmuille on kuitenkin kyettävä määrittelemään jollakin muulla tavalla niiden arvo, ts. kuinka suotuisia ne ovat pelaajan kannalta. Tähän arviointiin käytettävää menetelmää kutsutaan *evaluaatiofunktioksi*. Evaluaatiofunktio arvioi solmun pelitilasta joillakin kriteereillä sen suotuisuuden, ja palauttaa sen arvona. Tällaisen evaluaatiofunktion onnistunut laatiminen vaatii *aluetietämystä* (domain knowledge), eli useimmiten korkean tason ihmispelaajien pelitietämyksen formalisointia ohjelmallisesti käsiteltävään muotoon. Tällaiset evaluaatiofunktiot on luonnollisestikin laadittava jokaisen pelin tapauksessa erikseen.

2.1 Minimax-algoritmi

Vakiomuotoinen minimax-puuhakualgoritmi toimii seuraavasti: Oletetaan, että kilpailullisessa vuoropohjaisessa pelissä on kaksi pelaajaa, *min* ja *max*. Puun terminaalisolmuille annetaan arvo, joka ilmaisee kyseisen terminaalisolmun suotuisuutta max-pelaajan näkökulmasta. Muille kuin terminaalisolmuille annetaan minimax-arvo rekursiivisesti. Jos jollakin ei-terminaalisolmulla on max-pelaajan siirtovuoro, niin silloin solmulle annetaan arvoksi sen lapsisolmujen suurin arvo. Vastaavasti min-pelaajan siirtovuoron solmuille annetaan sen lapsisolmujen arvoista pienin arvo (kuva 1). (Campbell et al., 1983)



Kuva 1: Minimax-puurakenne

Kuten aiemmin on todettu, koko puun avaaminen terminaalisolmuihin asti ei useimmissa tapauksissa ole laskentatehollisesti mahdollista. Tällöin voidaan esimerkiksi syvyysrajatussa haussa käyttää syvimpien solmujen eli lehtisolmujen evaluaatiofunktion arvoa terminaaliarvojen sijasta.

Syvyysrajoitetun minimax-algoritmin toiminta pseudokoodina (Russel ja Norvig, 2016):

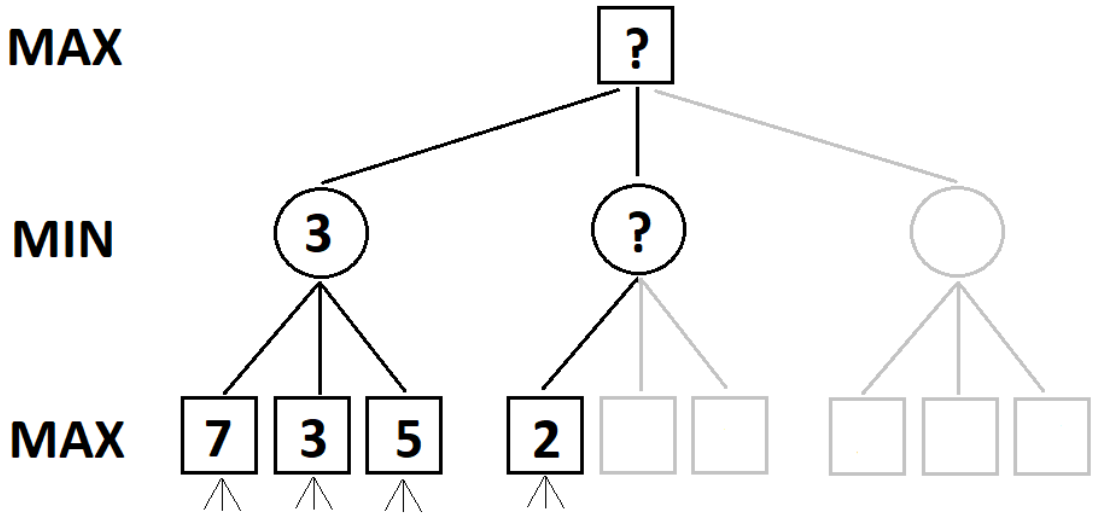
```
function minimax(node, depth, maxPlayer) is
    if depth = 0 or node is terminal then
        return value of the node
    if maxPlayer then
        value = −∞
        for each child node of node do
            value = max(value, minimax(child, depth−1, false))
        return value
    else
        value = +∞
        for each child node of node do
            value = min(value, minimax(child, depth−1, true))
        return value
```

2.1.1 Alfa-beta-karsiminen

Hakupuun syvyyttä rajaamalla voidaan estää minimax-puuhaun laajeneminen laskennallisesti mahdottoman suureksi käsitellä, mutta tämä tapahtuu luonnollisestikin tekoälyn toiminnan laadun kustannuksella. Jotta puuhaun syvyyttä voitaisiin kasvattaa tai toteutuksen nopeutta lisätä, on usein tarvetta tehostaa minimax-algoritmin toimintaa. Yksi laajalti käytetty keino tähän on *alfa-beta-karsiminen* (alpha-beta pruning). Alfa-beta-karsimisessa tunnistetaan sellaiset hakupuun haarat, joilla ei ole merkitystä minimax-algoritmin toiminnan kannalta, ja jätetään ne kokonaan tutkimatta (Fuller et al., 1973).

Alfa-beta-karsiminen toimii seuraavasti: Minimax-algoritmin toimintaan lisätään *alfa*- ja *beta-arvot*, joista alfa-arvon alkuarvo on miinus ääretön ja beta-arvon arvo puolestaan ääretön. Kun puun solmuja käydään läpi, verrataan niiden saamia arvoja alfa- ja beta-arvoihin siten, että max-pelaajan arvoja verrataan alfaan ja min-pelaajan arvoja betaan. Mikäli solmun arvo ensimmäisessä tapauksessa on suurempi kuin alfan arvo, se sijoitetaan alfaan, ja jälkimmäisessä tapauksessa, solmun arvon ollessa betaa pienempi, se sijoitetaan betaan. Tällöin kyseisen solmun lapsisolmuja läpi käytäessä tiedetään, että mikäli lapsisolmun arvo on pienempi kuin alfa-arvo tai vastaavasti suurempi kuin beta-arvo, ei kyseisen solmun lapsia kannata käydä läpi enempää. (Fuller et al., 1973)

Kuvassa 2 on havainnollistettuna tilanne, jossa ensimmäisen puuhaaran arvoksi min-pelaajan tasolla on saatu 3. Keskimmäistä haaraa tutkiessa sen ensimmäinen lapsisolmu saa arvon 2. Tällöin tiedämme, että keskimmäisen haaran min-tason solmun arvoksi tulee *enintään* 2, jolloin ylimmän tason max-pelaaja ei koskaan valitse sitä. Sen jatkotutkiminen olisi siis turhaa, ja algoritmi voi suoraan siirtyä tutkimaan oikeanpuoleista haaraa.



Kuva 2: Tilanne, jossa alpha-beta-karsinta ei tutki keskimmäistä haaraa enempää

Alfa-beta-karsimisen toiminta pseudokoodina (Russel ja Norvig, 2016):

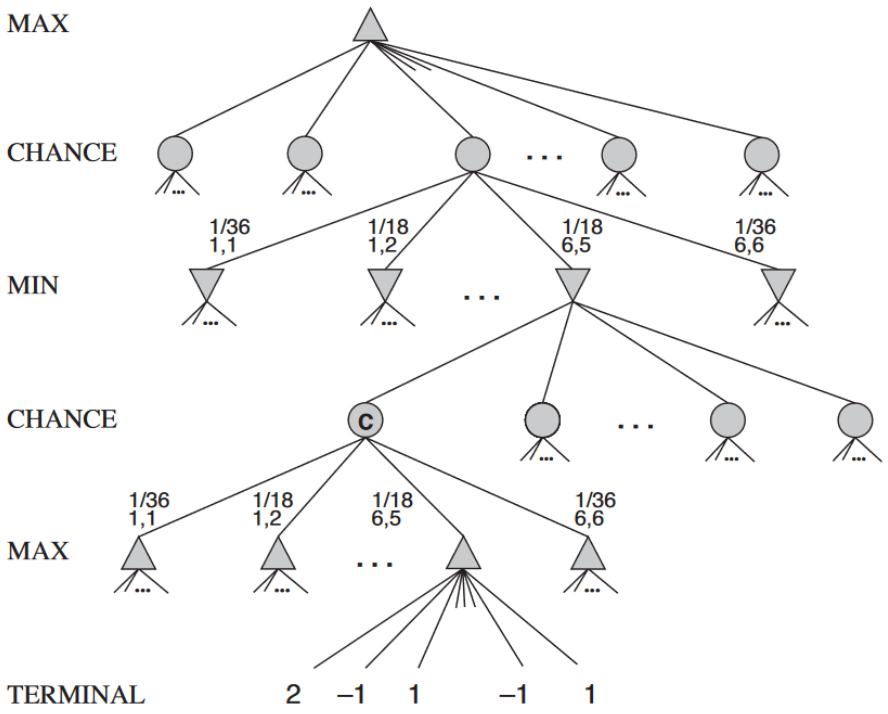
```
function alphabeta(node, depth, a, b, maxPlayer) is

    if depth == 0 or node is a terminal node then
        return evaluated value of the node
    if maxPlayer then
        value = −∞
        for each child node of node do
            value = max(value, alphabeta(child, depth−1, a, b, FALSE))
            if value ≥ b then
                break
            a = max(a, value)
        return value
    else
        value = +∞
        for each child node of node do
            value = min(value, alphabeta(child, depth−1, a, b, TRUE))
            if value ≤ a then
                break
            b = min(b, value)
        return value
```

Alfa-beta-karsimisella saavutetaan sama tulos kuin vakiomuotoisella minimaxilla, mutta vältetään paljon turhaa puun avaamista, jolloin puuta voidaan tutkia joidenkin havaintojen mukaan jopa kaksi kertaa syvemmälle kuin pelkällä naiivilla minimaxilla. (Fuller et al., 1973)

2.1.2 Expectiminimax

Minimaxin optimaalinen toiminta olettaa, että kumpikin pelaaja suorittaa jokaisessa tilanteessa omasta näkökulmastaan parhaan mahdollisen toiminnon, eikä valintoihin liity stokastisuutta. Satunnaisuutta sisältävissä pelissä tilanne ei kuitenkaan useimmiten ole edellä kuvatun kaltainen, vaan esimerkiksi nopanheitto saattaa vaikuttaa pelin etenemiseen monellakin tapaa, kuten rajaamalla pelaajan laillisten siirtovaihtoehtojen määrää.

Tällaisissa tilanteissa voidaan päästä parempiin tuloksiin minimax-algoritmin *expectiminimax*-muunnelmalla. Expectiminimaxissa pelipuuhun sisällytetään min- ja max-solmujen lisäksi myös *mahdollisuussolmuja* (chance nodes). Kuvassa 3 mahdollisuussolmujen lapset edustavat jonkin satunnaistekijän kaikkia mahdollisia lopputuloksia, kuten esimerkiksi nopan silmälukuja. Tällaisen solmun arvoksi lasketaan yleensä kaikkien sen lapsisolmujen keskiarvo. (Russel ja Norvig, 2016)



Kuva 3: Expectiminimax-puurakenne (Russel ja Norvig, 2016)

Expectimax-variantin toiminta pseudokoodina (Russel ja Norvig, 2016):

```
function expectimax(node, depth)

    if node is terminal or depth = 0
        return evaluated value of the node

    if opponent is to play
        // return value of minimum child
        let a = +∞
        foreach child node of node
            a = min(a, expectimax(child, depth-1))

    else if player is to play
        // return value of maximum child
        let a = -∞
        foreach child node of node
            a = max(a, expectimax(child, depth-1))

    else if randomness at node
        // return average of all child values
        let a = 0
        foreach child node of node
            a = a + (probability[child] X expectimax(child, depth-1))

    return a
```

2.3 Monte Carlo -puuhaku (MCTS)

Monte Carlo -puuhaku suorittaa hakupuussa halutun kokoisen sarjan simuloituja läpipeluita, joissa pelitoimintojen valinnat tehdään satunnaisesti kaikkien laillisten toimintojen joukosta. Jokaisen läpipeluun eli simulaation jälkeen kaikki kyseisessä simulaatiossa läpikäydyt solmut terminaalisolmun ja kyseisen simulaation juurisolmun välillä päivitetään terminaalisolmun arvolla. Toistamalla tätä inkrementaalisesti kerätään hakupuusta ikään kuin tilastollisia näytteitä, ja näin saatua tilastollista dataa siirtojen suotuisuudesta voidaan käyttää toiminnon valitsemisessa tai tekoälyagentin jatkotoiminnan ohjaamisessa hyväksi.

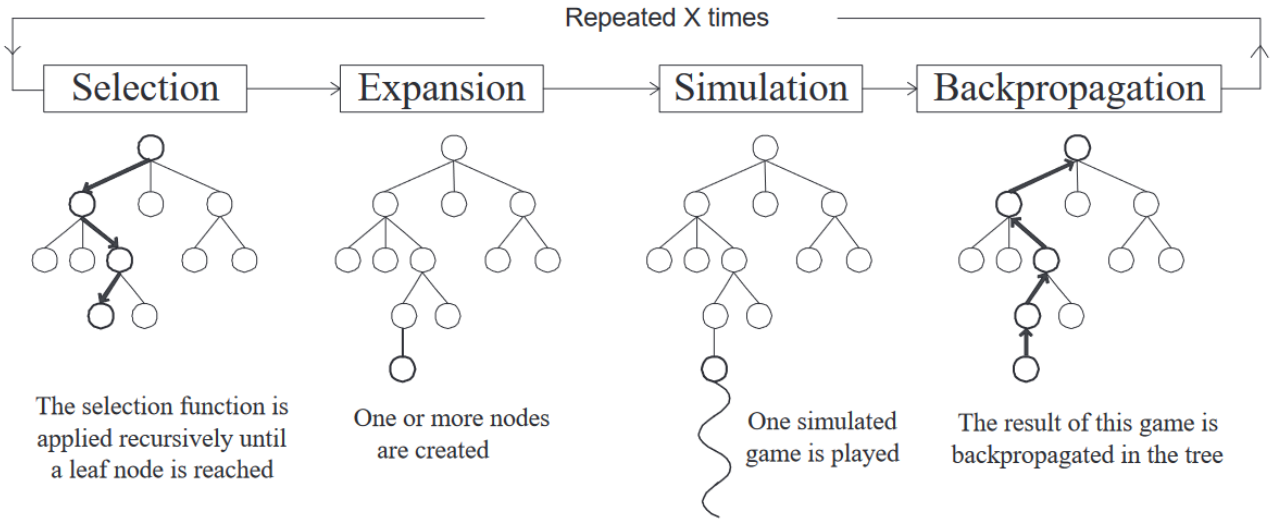
Sanallisesti kuvattuna vakiomuotoisen Monte Carlo -puuhaun jokaisen kierroksen toiminta (kuva 4) voidaan jakaa seuraaviin neljään vaiheeseen (Chaslot et al., 2008):

1. *Valinta* (selection): Alkaen juuresta J, joka vastaa tämänhetkistä pelitilaa, valitaan perättäisiä lapsisolmuja kunnes päästään lehtisolmuun L, joka on solmu, jolla on potentiaalisia lapsisolmuja joista ei ole vielä aloitettu simulaatiota.

2. *Laajennus* (expansion): Ellei L ole terminaalisolmu, laajennetaan yksi tai useampi lapsisolmu ja valitaan niistä solmu S.

3. *Simulaatio* (simulation): Suoritetaan yksi satunnainen pelin läpipeluu terminaalisolmuun asti (simulaatio) alkaen solmusta S. Yksinkertaisimmillaan tämän simulaation polku valitaan satunnaisesti.

4. *Takaisinpropagaatio* (backpropagation): Läpipeluun tuloksella päivitetään kaikkien solmujen tiedot polulla solmusta S solmuun J.



Kuva 4: Monte-Carlo-puuhaku (Chaslot et al., 2008)

2.3.1 Upper Confidence bounds applied to Trees -menetelmä

Yksi Monte Carlo -puuhaun ongelmista on tilanne, jossa jokin siirto saa simulaatioiden tuloksena korkean arvon silloin, jos valtaosalla sen jatkosiirroista saavutetaan hyviä arvoja, mutta yksi tai useampi näistä jatkosiirroista johtaa hävittyyn tai muuten erittäin epäsuotuisaan tilanteeseen (Kocsis ja Szepesvári, 2006). Esimerkiksi shakkipeleissä hyvin yleinen tilanne on, että muutoin voittavassa asemassa, jossa valtaosa jatkosiirroista pitää tilanteen hyvänä, vastustajalla on kuitenkin yksi siirto, joka kaataa koko pelin ja kääntää aseman päälaelleen. Vastaavasti jossakin tilanteessa valtaosa siirroista saattaa olla pelaajalle epäsuotuisia, mutta jokin tietty siirto tai siirtosarja tuottaisi valtavan

edun. Jos MCTS-simulaatioissa keskitytään liiaksi tutkimaan jo hyväksi todettuja puun haaroja, saattavat tällaiset siirrot jäädä löytymättä.

Ratkaisuksi edellä mainittuun ongelmaan on käytetty MCTS:n *UCT-muunnelmaa* (Upper Confidence bounds applied to Trees). Tällöin simulaatioissa valittuja solmuja ei valita täysin satunnaisesti, vaan painotetaan UCT-evaluaatiolla (kaava 1), jolloin valintaa kohdellaan *monikätisen hedelmäpelin ongelmana* (multi-armed bandit problem). UCT:lla pyritään löytämään sopiva tasapaino jo hyväksi todettujen sekä uusien, vielä kohtalaisen tuntemattomien haarojen tutkimisen välillä, mikä on yksi MCTS-algoritmin käytön suurimmista haasteista. (Kocsis ja Szepesvári, 2006)

$$\left(\frac{w_i}{n_i}\right) + c\sqrt{\frac{\ln\left(N_i\right)}{n_i}}$$

Kaava 1: UCT:n kaava (Kocsis ja Szepesvári, 2006)

UCT:ssa valitaan lapsisolmu *s*, joka maksimoi kaavan 1 UCT-evaluaation. Kaavassa w_i on solmun voittojen määrä, n_i solmun simulaatioiden määrä, N_i solmun vanhemman simulaatioiden määrä ja c tutkimusvakio (exploration parameter), jonka suuruus valitaan toteutuskohtaisesti. Vakiota säätämällä voidaan kannustaa simulaatiototeutusta tutkimaan toistaiseksi tuntemattomampia puun haaroja jo hyväksi todettujen lisätutkimisen kustannuksella. (Kocsis ja Szepesvári, 2006)

2.3.2 Rapid Action Value Estimation -menetelmä

Vakiomuotoinen MCTS joutuu useimmiten tutkimaan puuta erittäin paljon, eli suorittamaan useita simulaatioita, ennen kuin se on kerännyt tarpeeksi tilastollista dataa parhaan siirron määrittelemiseksi riittävällä varmuudella. Joissakin toteutuksissa tätä tutkimusaikaa voidaan kuitenkin merkittävästi vähentää *RAVE-muunnelmaa* (Rapid Action Value Estimation) käyttämällä. RAVE:n hyödyntämiseen soveltuvat erityisesti pelit, joissa siirtosarjojen permutaatiot usein johtavat samaan peliasemaan, kuten esimerkiksi go-pelissä. (Gelly ja Silver, 2011)

RAVE käyttää toiminnassaan AMAF-heuristiikkaa (All Moves As First). AMAF-heuristiikassa

kaikki samat siirrot jakavat keskenään saman arvon, riippumatta siitä, missä kohtaa hakupuuta ne esiintyvät. RAVE:ssa solmujen sisältöön eivät siis simulaatioita suorittaessa vaikuta ainoastaan välittömästi solmussa pelatut pelitoiminnot, vaan myöskin samat toiminnot myöhemmin pelattuna. RAVE voi merkittävästi nopeuttaa Monte Carlo -puuhaun toimintaa simulaatioiden alkuvaiheessa, joskin oikeellisuuden kustannuksella. (Gelly ja Silver, 2011)

3. Tutkimusmenetelmä

Tämän tutkielman tarkoitus on kartoittaa MCTS-algoritmin soveltuvuutta ja käyttöä vuoropohjaisten strategiapelien tekoälyratkaisuissa pohjautuen olemassa olevaan tieteelliseen kirjallisuuteen. Tutkimusmenetelmänä on kirjallisuuskatsaus, joka toteutettiin soveltaen, joskin löyhästi, systemaattisen kirjallisuuskatsauksen ohjenuoria (Kitchenham ja Charters, 2007) . Algoritmien kuvauksiin liittyvä kirjallisuushaku toteutettiin enimmäkseen vapaamuotoisena hakuna tutkielman tarpeisiin soveltuvasti. Kirjallisuushaku koskien tässä tutkielmassa tarkasteltujen vuoropohjaisten strategiapelien tekoälyratkaisuja sekä MCTS-algoritmin käyttöä toteutettiin systemaattisemmin, jotta varmistetaan riittävä kattavuus.

Aineiston valikoitumisen perusteena tutkielman lähdemateriaaliksi käytettiin potentiaalisen lähteen relevanssia tutkielman aiheen kannalta. Relevanssikriteerinä toimi se, käsitteleekö aineisto pelin tekoälyratkaisuja tai MCTS-algoritmin soveltamista johonkin neljästä pääasiallisen tarkastelun kohteena olevasta pelistä, tai johonkin muuhun vuoropohjaiseen strategiapeliin, jollakin mielenkiintoisella tavalla. Lähdeaineistoksi priorisoitiin etenkin sellaisia lähteitä, jotka keskittyivät erityisesti MCTS-algoritmin ja sen varianttien soveltamiseen joko sellaisenaan tai yhdessä joidenkin muiden tekoälytekniikoiden kanssa. Muiden kuin neljän pääasiallisen tarkastelun kohteena olevan pelin osalta priorisoitiin sellaisia lähteitä, jotka käsittelivät MCTS-algoritmia ja sen hyödyntämistä jonkin vähintäänkin kohtalaisen tunnetun ja laajalti pelatun vuoropohjaisen strategiapelin kohdalla.

Kirjallisuushaun tavoitteena oli löytää yhteensä n. 20 relevanttia ja riittävän laadukasta lähdettä neljän pääasiallisen tarkastelun kohteena olevan pelin osalta, ja lisäksi n. 5 lähdettä muiden vuoropohjaisten strategiapelien osalta. Tarkasteltavaan aineistoon pyrittiin rajaamaan jokaisen pelin kohdalla muutamat eniten relevantit lähteet, jotta aineiston kokonaismäärä pysyisi kohtuullisena. Näin toimittiin siksi, että tutkielman tarkoitus ei ole keskittyä mihinkään yhteen peliin erityisen laajasti, vaan luoda yleisluontoisempi katsaus tarkastelemalla ratkaisuja useammassa pelissä. Aineistoa luonnollisestikin laajennettiin myös tarkastelemalla relevantin aineiston lähdeluetteloita. Lopulta valikoidun aineiston kokonaismäärä koostui neljän laajemmin tarkastellun pelin osalta 19 lähteestä, jakautuen lähes tasaisesti kaikkiin peleihin. Lisäksi mukaan sisällytettiin yksi mielenkiintoinen MCTS-puuhakua käsittelevä lähde kustakin seuraavista peleistä: *Hearthstone*, S*ettlers of Catan*, *Carcassonne*, *Diplomacy* ja *Spades*.

4 Tarkasteltavat strategiapelit

Tässä luvussa tarkastellaan tekoälykontekstissa lähemmin neljää strategiapeliä (shakki, go, pokeri ja Magic: The Gathering) ja aiemmin tarkasteltujen tekoälymenetelmien soveltuvuutta kyseisiin peleihin olemassa olevaan kirjallisuuteen perustuen. Kaikki neljä peliä ovat vuoropohjaisia, ja lähes kaikki pokeria lukuun ottamatta pääsääntöisesti (ja tämän tutkielman rajauksen puitteissa) kahden pelaajan kilpailullisia pelejä, jolloin tässä tutkielmassa tarkasteltujen puuhaku-algoritmien voidaan katsoa lähtökohtaisesti soveltuvan hyvin kyseisten pelien tekoälyratkaisuihin. Lisäksi pelit eroavat toisistaan monelta tekoälyn kannalta merkittäviltä ominaisuudeltaan, kuten täydellisen tiedon, haarautumiskertoimen ja stokastisuuden suhteen, ja tarjoavat sikäli mielenkiintoisen heijastuspinnan etenkin Monte Carlo -puuhaun soveltuvuudelle.

4.1 Shakki

Tässä alaluvussa kuvataan lyhyesti, joskaan ei kattavasti, shakin säännöt ja shakkitekoälyn haasteet, sekä tutustutaan perinteistä puuhakuratkaisua ja ihmispelaajien tietämykseen perustuvaa asema-evaluaatiota hyödyntävään Stockfish-shakkimoottoriin. Tämän jälkeen tutustutaan shakkitekoälyn kannalta olennaisiin tai peliin sovellettuihin MCTS-ratkaisuihin.

4.1.1 Shakin kuvaus

Shakki on vanha ja hyvin tunnettu kahden pelaajan vuoropohjainen strategiapeli, joka on etenkin viime vuosina kokenut eräänlaisen laajamittaisen renessanssin erinäisten siihen liittyvien populaarikulttuurin ilmiöiden ja mediatuotosten seurauksena. Peliä pelataan 8x8 ruudun laudalla, ja kullakin pelaajalla on käytössään 16 nappulaa, jotka eroavat liikkuvuudeltaan toisistaan. Pelaajat ovat nimeltään *musta* ja *valkea*, mikä heijastuu yleensä nappuloiden fyysisessä värissä. Valkea pelaaja aloittaa, ja pelaajat siirtävät kukin vuorollaan yhtä nappulaa (*siirto*), ja nappulan siirtyessä ruutuun, jossa sijaitsee vastustajan nappula, tulee kyseinen vastustajan nappula *syödyksi* ja poistuu pelilaudalta pysyvästi. Kullakin pelaajalla on yksi kuningasnappula, ja pelin voittaa se pelaaja, joka ensimmäisenä uhkaa syödä vastustajan kuninkaan (*shakki*) siten, että vastustaja ei millään omalla laillisella siirrollaan kykene estämään kuningastaan tulemasta syödyksi (*shakkimatti*).

4.1.2 Tekoälyn haasteet shakissa

Shakkitekoälyä on tutkittu ja kehitetty jo pitkään. Ehkäpä tunnetuin ja laajinta maailmanlaajuista huomiota saanut shakkitekoälyn riemuvoitto oli, kun IBM:n shakkitietokone *Deep Blue* voitti

silloisen maailmanmestari Garry Kasparovin kuuden pelin ottelussa vuonna 1997 (Campbell et al., 2002). Shakin kussakin pelitilanteessa laillisten siirtojen keskimääräisesti korkea määrä ja siitä seuraava haarautumiskerroin (n. 35) aiheuttavat sen, että koko pelipuuta ei ole läheskään mahdollista tutkia vielä nykyäänkään saatavilla olevalla laskentateholla. Nk. *Shannonin luku* (10^{120}) on konservatiivinen alaraja-arvio shakin täydellisen pelipuun koolle, joten on selvää, että täydellisen optimaalista tekoälyratkaisua ei voitane shakille ainakaan lähitulevaisuudessa toteuttaa (Maharaj et al., 2022). Deep Blue:n aikana syvyysrajoitukset olivat laskentatehollisista syistä vielä huomattavasti nykyistä ankarammat, vaikka kone käyttikin rinnakkaisuutta tehokkaasti hyväkseen laskennassaan. Deep Blue hyödynsikin toiminnassaan sekä ihmissuurmestarien peleistä koottua tietokantaa että monimutkaista evaluaatiofunktiota hyvien siirtojen löytämiseen (Campbell et al., 2002).

Koska shakki on täydellisen informaation peli eikä sisällä mitään stokastisia elementtejä, voidaan siihen sovellettavien tekoälymenetelmien haasteena pitää ensisijaisesti aiemmin mainittua korkeaa haarautumiskerrointa ja siitä seuraavaa erittäin suurta puurakennetta.

4.1.3 Shakkiin sovellettuja tekoälymenetelmiä - Stockfish

Nykyään tunnetuin ja laajimmin käytössä oleva shakkitekoälyn toteutus on *Stockfish*-shakkimoottori. Alun perin 2008 julkaistu ilmainen ja avoimen lähdekoodin Stockfish on hallinnut shakkitekoälymaailmaa jo pitkään, ja puolustanut shakkitekoälykilpailuissa asemaansa onnistuneesti uudempia kilpailijoita vastaan, joista yhteen haastavimmista tutustutaan aliluvussa 3.1.4 tarkemmin. Parhaatkin ihmispelaajat Stockfish on jo jättänyt kauas taakseen.

Stockfish perustuu aiemmin kohdassa 2.1.1 tarkasteltuun minimax-algoritmin alfa-beta-karsimisvarianttiin syvyysrajoitettuna. Syvyysrajoitus on Stockfishin tapauksessa toteutettu hyödyntäen *iteratiivisena syventämisenä* (Iterative Deepening Depth First Search, IDDFS) tunnettua tekniikkaa. Iteratiivisessa syventämisessä syvyyshakua estetään menemästä määriteltyä rajaa syvemmälle, ja tätä syvyysrajaa voidaan inkrementaalisesti kasvattaa. Stockfishin tapauksessa ilmoitettu syvyysraja ei kuitenkaan tarkoita, että haku olisi käynyt läpi kaikki mahdolliset pelivariaatiot tällä syvyydellä, sillä Stockfishin heuristiikka ohjaa hakua tutkimaan lupaavia variaatioita rajaa syvemmälle ja vähemmän lupaavia variaatioita matalammalle. (Maharaj et al., 2022)

Stockfish-shakkimoottori hyödyntää kahta heuristiikkaluokkaa tutkittavan hakupuun pienentämiseen. Ensimmäinen näistä on *edestäkarsiminen* (forward pruning). Edestäkarsimistekniikoissa, hieman luvussa 2 tarkastellun alfa-beta-karsimisen tavoin, karsitaan pois tarkastelusta haaroja, jotka luultavasti eivät sisälly optimaaliseen peliin. Jos jonkin peliaseman eli puun solmun evaluaatiofunktion arvo on huomattavasti huonompi kuin pelaajan paras vaihtoehto, kyseisen solmun lapsisolmut voidaan karsia pois. Toisessa heuristiikkaluokassa, *reduktioheuristiikassa,* joitakin puun haaroja tutkitaan matalammin, sen sijaan että ne karsittaisiin kokonaan pois. (Maharaj et al., 2022)

Lehtisolmujen arvioinnissa käytettävä evaluaatiofunktio arvioi Stockfishin aiempien versioiden (versio 11 ja aiemmat) tapauksessa peliasemaa perustuen ennalta määriteltyihin peliaseman ominaisuuksiin, kuten materiaalin määrään eli laudalla olevien nappuloiden määritettyyn arvoon, niiden erotukseen pelaajien välillä sekä nappuloiden suhteellisten sijaintien mahdollistamaan *aktiviteetti*potentiaaliin suhteutettuna pelin senhetkiseen *pelivaiheeseen*, jotka shakkipelin tapauksessa jaotellaan karkeasti *alku*-, *keski*- ja *loppupeliin*. Uudemmissa versioissa Stockfishiin on toteutettu neuroverkkoa hyödyntävä evaluaatiofunktio, joka on koulutettu ennakoimaan klassisen evaluaatiofunktion palautusarvoa. (Maharaj et al., 2022)

4.1.4 MCTS:n soveltaminen shakin tekoälyyn

Ansatilat (trap states) ovat laajalti tutkittu ilmiö. Ansatiloissa jokin pelin siirto vaikuttaa lyhyellä tähtäimellä pelaajan kannalta edulliselta, mutta osoittautuukin pelin jatkuessa haitalliseksi, jopa tuhoisaksi. Companezin ja Aletin (2016) mukaan ansatilojen vastapareja, *uhraustiloja* (sacrifice states) on kuitenkin tutkittu huomattavasti vähemmän. Uhraustilat ovat sellaisia pelin tiloja, joissa jokin siirto saattaa aiheuttaa lyhyellä tähtäimellä peliaseman merkittävääkin heikentymistä, mutta osoittautuu kuitenkin pitkällä tähtäimellä edulliseksi. Uhraussiirroille on tyypillistä, että uhrauksesta saatavan edun realisoiminen vaatii tietyn siirtosarjan tai -sarjojen seuraamista. Muussa tapauksessa uhrauksen hyöty menetetään. Etenkin shakissa tällaiset tilanteet ovat verrattain yleisiä, kuten jokainen hiemankaan kokeneempi shakinpelaaja luultavasti tietää. Monet maineikkaat shakinpelaajat, kuten esimerkiksi ”Riikan taikuri” Mikhail Tal, niittivät aikoinaan mainetta erityisesti näyttävillä uhraussiirroillaan, ja hänen pelejään analysoidaan vielä nykypäivänäkin.

Companez ja Aleti (2016) tutkivat MCTS-algoritmin mahdollisuuksia parantaa toimintaansa edellä kuvatun kaltaisissa ansatilanteissa hyödyntäen kolmea eri MCTS-varianttia: *Ratkaisevien siirtojen* (decisive moves) varianttia, *painotettujen päivityksien* (weighted updates) varianttia ja aiemmin

kuvattua RAVE-varianttia. Kokeensa he suorittivat *ultimate tic-tac-toe* -ristinollapelimuunnelmalla, koska se mahdollisti sellaisten suoraviivaisten skenaarioiden luonnin, jossa toinen pelaaja tekee pakottavia siirtoja, jotka ovat yksi tavallisimmista uhraustyypeistä. Tulosten voidaan silti katsoa olevan pääpiirteiltään sovellettavissa shakkiinkin.

Ratkaisevat siirrot -variantin toiminta poikkeaa vakio-MCTS:n toiminnasta siten, että satunnaisvalinnan sijaan tarkastetaan ensin, onko siirtovaihtoehtoissa pelaajalle välittömästi voittavaa siirtoa. Mikäli sellainen löytyy, se valitaan, mutta muussa tapauksessa suoritetaan satunnaisvalinta. Vaikka tällainen tarkistus vaatikin oman laskennallisen hintansa, se kuitenkin lyhentää keskimääräistä simulaation pituutta. (Companez ja Aleti, 2016)

Xien ja Liun (2009) kehittämässä painotetut päivitykset -variantissa puolestaan muokataan algoritmin takaisinpropagaatiovaihetta. Sen sijaan että voitto pisteytettäisiin arvolla 1, häviö arvolla 0 ja tasapeli arvolla 0,5, arvot painotetaan sen mukaan, kuinka monta siirtoa lopputulokseen johtavassa pelissä tehtiin. Mitä enemmän pelissä tehtiin siirtoja, sitä heikomman painotuksen lopputuloksen arvo saa takaisinpropagaatiossa. Tällaisen painotuksen seurauksena korostuvat erot kahdenlaisten peliasemien välillä: Niiden, joilla on taipumusta johtaa nopeisiin voittoihin ja niiden, jotka yleensä johtavat pidempiin peleihin. Kolmannen variantin eli RAVE:n toimintaa on selitetty tämän tutkielman aliluvussa 2.3.2. (Companez ja Aleti, 2016)

Companez ja Aleti (2016) havaitsivat, että ratkaiseva-variantti suoriutui kolmesta muunnelmasta parhaiten sekä terminaalisissa että ei-terminaalisissa skenaarioissa. Terminaaliset skenaariot ovat pelitilanteita, joissa uhrauksella saavutettava hyöty on suora pelin voitto, kun taas ei-terminaalisissa skenaarioissa hyöty on asemallinen, ei suoraan voittava. Ratkaiseva-variantin käyttämä voittotilanteiden etsintä jokaisessa solmussa vaatii toki ylimääräistä laskentaa, mutta koepelin kohdalla lisätaakka ei ollut merkittävä. Ylimääräisen laskentataakan suuruus on toki pelikohtainen, mutta useimmissa peleissä, kuten shakissa, sen voidaan kuitenkin katsoa todennäköisesti pysyvän melko kevyenä.

Stockfishin varteenotettavaksi kilpailijaksi nousi vuonna 2017 julkaistu DeepMind:in *AlphaZero*, joka nostatti kohua aiemmin lähinnä Stockfishin hallitsemissa shakkipiireissä. AlphaZero korvaa perinteisemmissä tekoälytekniikoissa, kuten Stockfishissä, käytetyn käsin rakennetun pelitietämyksen ja muut pelikohtaiset tekoälyominaisuudet syvällä *neuroverkolla*, yleiskäyttöisellä *vahvistusoppimisalgoritmilla* sekä yleiskäyttöisellä puuhakualgoritmilla, tässä tapauksessa Monte Carlo -puuhaulla. Näitä kolmea tekniikkaa yhdistäen AlphaZero pelaa itseään vastaan käyttäen

ainoastaan pelin sääntöjä pohjanaan, ja kouluttaa näin itseään yhä suorituskykyisemmäksi. (Silver et al., 2018)

Alfa-beta-karsintaa käyttävän puuhaun sijasta AlphaZero hyödyntää MCTS:ää, suorittaen pelipuussa sarjan simulaatioita juuresta lehtisolmuihin asti. Jokaisessa simulaatiossa valitaan jokaisesta pelitilasta sellainen siirto, joka on verrattain harvoin valittu aikaisemmissa simulaatioissa ja jolla on neuroverkon mukaan aiempiin simulointeihin perustuen korkea arvo. AlphaZeron neuroverkko siis käytännössä ohjaa MCTS-algoritmin toimintaa. (Silver et al., 2018)

Peleissään Stockfishiä vastaan AlphaZero osoittautui hallitsevaksi, voittaen 155 ja häviten vain 6 pelatuista tuhannesta pelistä. AlphaZeroa pelautettiin Stockfishiä vastaan myös siten, että pelit aloitettiin yleisistä ihmispelaajien käyttämistä avaustilanteista. AlphaZero oli näissäkin peleissä voitokas jokaisen avauksen kohdalla, vaikka toisin kuin Stockfishillä, sillä ei ollut käsin rakennettua aluetietämystä kyseisistä avauksista. Sen sijaan se oli itsenäisesti löytänyt näitä avauksia pelatessaan itseään vastaan. Lisäotteluita pelautettiin myös Stockfishin silloista uudempaa kehitysversiota sekä vahvaa avauskirjatietämystä käyttävää muunnelmaa vastaan. Näissäkin peleissä AlphaZero voitti kaikki ottelut. AlphaZero osoitti myös olevansa sääntöpohjaista Stockfishia kykenevämpi aiemmin käsiteltyyn uhrauspeliin, sillä useissa peleissä se uhrasi materiaalia saavuttaakseen pitkän aikavälin strategista etua. (Silver et al., 2018)

Yleisesti nähtiin, että shakin kohdalla jo MTCS-puuhakuratkaisuja edeltävä perinteisempi minimax- ja evaluaatiopohjainen Stockfish-shakkimoottori kykeni haastamaan ja voittamaan korkeimman tason ihmispelaajat. Shakin tekoälykehityksessä ei siis voida katsoa MCTS-tekniikoiden aiheuttaneen vallankumouksellista hyppyä toimijoiden pelivahvuudessa. AlphaZero kuitenkin osoitti, että MCTS-pohjainen yleisluontoinen ratkaisu kykenee kehittämään pelityylejä, jotka ovat perinteisemmille ratkaisuille epätyypillisiä, ja sekä haastamaan että jopa ylittämäänkin nämä ratkaisut pelivahvuudessa.

4.2 Go

Tässä alaluvussa kuvataan pääpiirteittäin go-pelin säännöt ja tekoälyn kannalta haasteelliset piirteet. Lisäksi tutustutaan go-tekoälyn perinteisempiä ihmistietämykseen pohjautuvia ratkaisuja hyödyntävään aiempaan tekoälykehitykseen, ja lopuksi tarkastellaan MCTS-pohjaisia ratkaisuja, joiden myötä go-tekoäly onnistui kehittymään huippuluokan ihmispelaajien tasolle, ja lopulta sen yli.

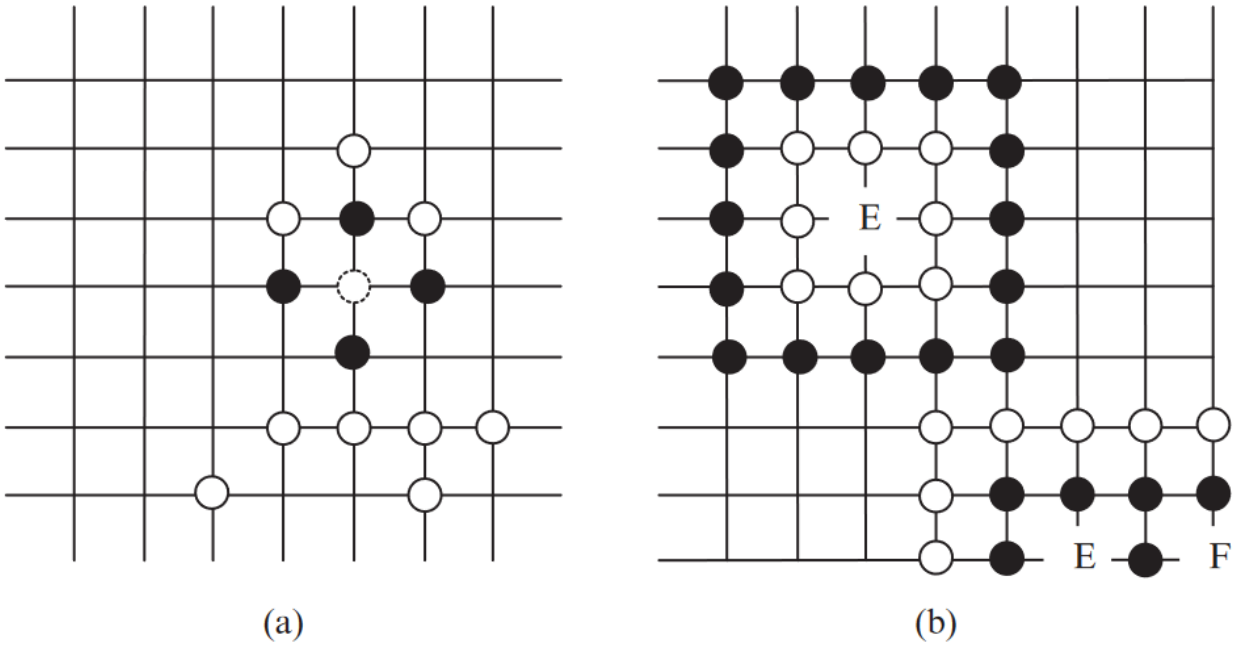
4.2.1 Go:n kuvaus

Go on vanha ja tunnettu lautapeli, joka kehitettiin Kiinassa yli 2 500 vuotta sitten, ja sen uskotaan olevan vanhin edelleen aktiivisesti pelattu lautapeli. Kuten shakki, go on kahden pelaajan kilpailullinen lautapeli, jossa pelaajat pelaavat kukin vuorollaan yhden siirron, ja shakin tavoin pelaajia kutsutaan *mustaksi* ja *valkeaksi*. Peli koostuu alussa tyhjästä laudasta joka sisältää 19x19 viivaruudukon, ja kukin pelaaja asettaa vuorollaan johonkin viivojen risteyskohtaan (*pisteeseen*) omaa väriään vastaavan nappulan, joita go:ssa kutsutaan *kiviksi*. Pelaaja voi myös halutessaan jättää vuoronsa väliin, eli *passata*. Jos molemmat pelaajat passaavat peräkkäin, peli päättyy. (Gelly et al., 2012)

Kivi tai kiviryhmä jää vangiksi eli poistetaan laudalta, jos sen suorakulmaisesti vierekkäisissä pisteissä on kaikissa vastustajan kivi, jolloin se menettää *vapautensa*, yhteytensä avoimiin pisteisiin. Kiveä ei myöskään saa asettaa paikkaan, jossa sillä ei olisi vapautta. Kiviryhmän sisällä olevaa tällaista pistettä kutsutaan *silmäksi*. Pelin päättyessä lasketaan pisteet, jotka määräytyvät sen mukaan, kuinka monta tyhjää pistettä pelaaja on onnistunut ympyröimään omilla kivillään, ja kuinka monta kiveä pelaaja on kaapannut. Laudan reunatkin toimivat pisteenlaskussa ympyröivänä elementtinä. (Gelly et al., 2012)

Kiellettyjä siirtoja ovat *itsetuhoiset* (suicidal) siirrot, joissa pelaaja asettaisi kivensä pisteeseen jossa sillä ei olisi vapautta - tällöin kivi tulisi saman tien kaapatuksi. Lisäksi pelaajat eivät saa asettaa nappulaansa siten, että asettaminen toistaisi aiemman laudan aseman. Tätä kutsutaan *Ko-säännöksi*, ja sen tarkoitus on estää peliä päätymästä loputtomaan kierteeseen. Kuvan 5 vasemmanpuoleinen asema on esimerkki Ko-säännöstä. Mustan siirto kaappaa valkoisen kiven poistamalla sen vapaudet. Valkoinen ei voi kuitenkaan asettaa kiveä uudelleen samaan pisteeseen, sillä silloin aiempi asema toistuisi. Kuvan 5 oikeanpuoleisessa asemassa valkoinen ei voi asettaa kiveä E-pisteeseen, sillä se poistaisi vapaudet koko kiviryhmältä. Musta voi asettaa kivensä E:hen, sillä vaikka pisteellä ei

olekaan vapauksia, siirto kaappaisi koko valkoisen kiviryhmän. Tässä tilanteessa valkoista kiviryhmää kutsutaan *kuolleeksi*. Muutoin pelin eteneminen on verrattain yksinkertaista - kukin pelaaja asettaa vuorollaan kiven pisteeseen, joka ei riko näitä kahta sääntöä. Go-pelistä on saatavana runsaasti laadukasta kirjallisuutta, joka kuvaa sen sääntöjä, strategioita ja pelin kulkua tarkemmin. (Cai ja Wunsch, 2007)



Kuva 5: Kaksi go-pelin asemaa (Cai ja Wunsch, 2007)

4.2.2 Tekoälyn haasteet go-pelissä

Go oli pitkään näennäisesti ylitsepääsemätön haaste tekoälyn saralla pelin rakenteesta johtuen. Laajasta pelilaudasta (361 pistettä) ja siirtojen vapaudesta (kiven saa asettaa mihin tahansa vapaaseen pisteeseen) johtuen go:n haarautumiskerroin on noin 250, jolloin perinteiset minimax-pohjaiset puuhakualgoritmit osoittautuvat hyvin nopeasti riittämättömiksi hakupuun laajentuessa laskennallisesti mahdottoman suuriksi käsitellä järkevässä ajassa (Cai ja Wunsch, 2007). Pelin *kombinatorinen kompleksisuus* (combinatorial complexity) on erittäin suuri, sillä jokaisella vuorolla on noin 200 mahdollista siirtoa, verrattuna esim. shakin 37:n. Lisäksi keskimääräinen pelin kesto vuoroissa on moninkertaisesti shakkia suurempi, ja mahdollisia pelitiloja on noin 10^{170}, siinä missä shakissa niitä on noin 10^{47} (Gelly et al., 2012). Go:n haastavuutta tekoälyratkaisuiden suhteen lisää myös se, että siirroilla voi olla erittäin kauaskantoisia vaikutuksia. Jo pelin alkupuolella tapahtuvalla kiven asettamisella saattaa olla valtaisa merkitys pelin lopputuleman kannalta satoja siirtoja

myöhemmin (Gelly et al., 2012).

Puuhaun syvyyden rajaaminen ei myöskään go:n tapauksessa tarjoa paljoakaan helpotusta silloin, kun tavoitteena on luoda peliä ammattilaistasolla pelaava tekoälytoimija. Lehtisolmujen suotuisuutta arvioiva evaluaatiofunktio on pelin luonteesta johtuen erittäin hankalaa luoda mielekkääksi, sillä ammattilaispelaajien tietämys on pääsääntöisesti intuitiopohjaista ja vaikeaa formalisoida koviksi säännöiksi. Tällöin esimerkiksi ihmispelaajien peleistä muodostettua tietokantaa ei voida shakin tavoin käyttää evaluaatiofunktion toiminnassa. Toisin kuin shakissa, ei go:ssa voi millään kovinkaan tehokkaalla tavalla arvioida materiaalin määrää tai aseman aktiivisuuspotentiaalia missään tietyssä peliasemassa, sillä kivillä ei ole minkäänlaista shakkinappuloita vastaavaa "arvoa". Kussakin tilanteessa jo ympyröityjen alueiden laskeminen shakkimateriaalia vastaavaksi materiaalieduksikaan ei ole go:n tapauksessa mielekästä, sillä muutamankin kiven asettaminen saattaa muuttaa tässä mielessä pelin tilannetta radikaalisti jo lyhyelläkin tähtäimellä, pidemmästä ajasta puhumattakaan. Lisäksi pelin päättyminen ei go:n tapauksessa ole pelin tilasta nähtävissä oleva asia kuten shakin ja shakkimatin tapauksessa, sillä peli päättyy vasta kun molemmat pelaajat passaavat peräkkäin. (Cai ja Wunsch, 2007)

4.2.3 Go-peliin sovellettuja tekoälymenetelmiä

Aikaiset pyrkimykset laatia tekoälyä go-peliin olivat pitkälti samankaltaisia kuin silloiset shakin vastaavat. Erittäin rajoittuneen käytettävissä olevan laskentatehon vuoksi puuhaku rajattiin vain pieneen osaan puuta, ja evaluaatiofunktiossa pelitiloja verrattiin staattisiin lautakuvioihin, jotka perustuivat ennalta rakennettuun korkean tason ihmispelaajatietämykseen kandidaattisiirtojen muodostamiseksi. (Cai ja Wunsch, 2007)

Yksi tällainen varhainen toteutus oli Zobristin (1969) ohjelma, jossa jokainen kivi "säteilee" ympäristöönsä etäisyyden myötä heikkenevän numeraalisen vaikutusalueen - mustan tapauksessa positiivisen ja valkoisen tapauksessa negatiivisen. Myös kiville itselleen annetaan numeraaliset arvot samaan tapaan. Tällöin jokaiselle laudan pisteelle voidaan laskea säteilyn mukainen numeraalinen arvo, ja näiden arvojen avulla tunnistaa mustan tai valkoisen pelaajan hallussa olevat laudan osat. Näin saatu data yhdistettiin muuhun aluetietämyksen pohjalta koostettuun dataan, kuten pisteiden tilaan, naapureiden määrään ja kiviketjujen kokoon, ja tämä koottiin sisäiseksi representaatioksi pelin tilasta. Tätä representaatiota verrattiin ihmispelaajien tietämystä sisältävään tietokantaan, joissa jokaisella pelin tilalla oli siihen liittyvä kandidaattisiirto tallennettuna. Ohjelma oli kuitenkin melko heikko, ja pärjäsi vain joitakin aloittelevia ihmispelaajia vastaan. (Cai ja

Wunsch, 2007)

Ryder (1971) laajensi Zobristin (1969) ohjelmaa hienostuneemmilla vaikutusaluefunktiolla ja laajemmalla pelitilan representaation piirteiden *summaevaluaatiolla*. Hän yhdisti strategiset, pitkän tähtäimen tavoitteet lyhyen tähtäimen taktiikoihin, sillä hän ymmärsi, että go edellyttää tasapainon säilyttämistä alueiden vahvistamisen ja tärkeiden kivien menettämisen välttämisen välillä. Ensin kandidaattisiirrot rajattiin summaevaluaatiolla noin 15 ehdokkaaseen, ja näitä siirtoja jatkoanalysoitiin sekä taktisesti että strategisesti. Tämän prosessin jälkeen valittiin pelattavaksi korkeimman pistemäärän saanut siirto. (Cai ja Wunsch, 2007)

1990-luvulla go-ohjelmien laadussa tapahtui huomattavaa kehitystä. Yksi hyvä esimerkki tällaisista ohjelmista on *Many Faces of Go*, joka nojasi raskaasti asiantuntijatietämykseen ja siihen pohjautuvaan sääntökokoelmaan. Evaluaatiofunktiossaan ohjelma ylläpiti dynaamista datarakennetta pelin tilasta, kuten pisteistä, kiviketjuista, yhteyksistä ja ryhmistä, ja muokkasi sitä inkrementaalisesti pelin edetessä. Pelimoottoria ajoi strategiafunktio, joka tutki tätä dynaamista datarakennetta valitakseen laudalta tärkeitä alueita siirtoja varten. Moottorista otettiin käyttöön eri versio pelin eri vaiheissa eli alku- keski- ja loppupelissä, joissa käytettiin eri tietämyspankkeja. Kaikkien tämäntyylisten vahvasti aluetietämykseen nojaavien ohjelmien pätevyys nojasi luonnollisestikin kyseisten aluetietämys-tietokantojen kattavuuteen kaikissa mahdollisissa tilanteissa, mikä on go-pelin tapauksessa vaikeaa, etenkin keskipelin osalta. (Cai ja Wunsch, 2007)

4.2.4 MCTS:n soveltaminen Go:n tekoälyyn

Brügmann (1993) kehitti ensimmäisen Monte Carlo -simulaatioita hyödyntävän go-tekoälyn nimeltään *Gobbel*, joka ei hyödynnä toiminnassaan juuri lainkaan ihmispelaajien aluetietämystä. Ohjelma on verrattain yksinkertainen, ja noudattaa Monte Carlo -simulaatioiden logiikkaa seuraavasti: Gobbel pelaa useita satoja pelejä kustakin asemasta lähtien siten, että jokainen siirto simulaatiossa valitaan satunnaisesti kaikkien asemassa laillisten siirtojen joukosta. Simulaatioissa Gobbel passaa vuoronsa ainoastaan silloin, kun tällaisia siirtoja ei enää ole suoritettavissa tai jos ainoa laillinen siirto laskisi jonkin sen kiviryhmän silmätilan kahdesta yhteen. Jälkimmäinen yksinkertainen tapaus on ainoa ohjelman toiminnassa hyödynnetty pelitietämyksen osa-alue.

Brügmann (1993) pelautti Gobbelia edellä kuvattua raskaasti aluetietämykseen nojaavaa Many Faces of Go -ohjelmaa vastaan. Lähes täydellisestä aluetietämyksen puutteestaan huolimatta jopa tämä erittäin varhainen simulaatiopohjainen toimija kykeni haastamaan Many Faces of Go -

ohjelman ja pelaamaan selvästi aloittelevaa ihmispelaajaa korkeammalla tasolla. Sen pelissä havaittiin kuitenkin huomattavia heikkouksia, kuten alueen puolustamisessa lähellä rajoja, mikä oli jo ajan monille muille ohjelmille helppo tehtävä. Lisäksi tässä varhaisessa vaiheessa myös käytettävän laskentatehon määrä luonnollisestikin rajoitti simulaatioiden määrää. (Brügmann, 1993)

Cazenave ja Helmstetter (2005) kehittivät *Monte Carlo -simulaatiot ja taktisen haun yhdistävää tekoälytoimijaa*. Heidän ensimmäisessä, vakiomuotoisessa toteutuksessaan Monte Carlo -puuhaku toimii seuraavasti: Ohjelma pelaa useita, yleensä vähintään tuhansia satunnaisia pelejä kustakin pelitilanteesta lähtien siten, että jokainen siirto simulaatiossa valitaan enemmän tai vähemmän satunnaisesti kaikkien asemassa laillisten siirtojen joukosta, kuitenkin niin, että siirto ei täytä pelaajan omaa silmää. Näissä simuloiduissa satunnaispelissä vuoro passataan ainoastaan silloin, kun tällaisia siirtoja ei enää ole suoritettavissa, ja pelin sääntöjen mukaisesti peli päättyy, kun kumpikin pelaaja passaa, jolloin suoritetaan pistelasku. Tämän jälkeen algoritmi laskee jokaiselle pisteelle keskiarvon niistä peleistä, joissa kyseinen siirto pelattiin ensimmäisenä, ja myös keskiarvon toiselle pelaajalle. Näiden arvojen erotuksesta muodostuu siirron arvo, ja algoritmi pelaa sen siirron, jossa tämä arvo on korkein.

Toisessa toteutuksessaan Cazenave ja Helmstetter (2005) yhdistävät viisi erilaista taktisen tavoitteen hakua MCTS-simulaatioihin. Nämä haut ovat *kaappaushaku*, jolla etsitään kaappaavaa siirtoa; *yhteyshaku*, joka etsii kiviketjuja, jotka voidaan yhdistää; *tyhjä yhteishaku*, joka pyrkii yhdistämään tyhjän pisteen kiviketjuun; *silmähaku*, joka etsii laudalta pisteitä jotka voisivat muodostaa silmän, sekä *"elämä ja kuolema" -haku*, jolla tarkastetaan kiviketjujen elinvoimaisuus. Näiden kertaalleen suoritettavien hakujen tuloksia käytetään valitsemaan ne tilastotiedot, joita MCTS-simulaatioilla kerätään.

Jälkimmäinen yhdistelmätoteutus osoittautui vakiomuotoista toteutusta vastaan pelatessa vahvemmaksi jopa silloin, kun edellisen suurempaa vuorokohtaista laskenta-aikaa kompensoitiin laskemalla simulaatioiden määrä 1 000:n, vakiomuotoisen suorittaessa 10 000 simulaatiota. Tällöin yhdistelmätoteutus oli vakiomuotoista kaksi kertaa nopeampi, mutta voitti silti keskimäärin 24.6 pisteellä, keskihajonnan ollessa 40 pistettä. Yhdistelmätoteutusta pelautettiin myös täsmälleen samoja taktisia hakuja käyttävää *Golois*-tekoälyä vastaan. Käsinlaadittuja heuristiikkoja Monte Carlo -simulaatioiden sijaan käyttävä Golois hävisi yhdistelmätoteutukselle keskimäärin 26 pisteellä, osoittaen Monte Carlo -simulaatioiden olevan lupaava vaihtoehto käsinlaaditulle evaluaatiolle. (Cazenave ja Helmstetter, 2005)

Vuonna 2006 Gelly et al. (2006) kehittivät tunnetun ja voitokkaan *MoGo*-nimisen go-ohjelman, joka oli ensimmäinen luvussa 2 kuvattua MCTS:n UCT-laajennusta hyödyntävä go-tekoäly. Myöhemmin Gelly ja Silver (2011) kehittivät MoGo:a edelleen kahdella laajennuksella: Luvussa 2 kuvatulla MCTS-RAVE -muunnelmalla ja heuristisella MCTS-haulla, joka käyttää heuristista funktiota alustamaan uusien pelitilojen arvon hakupuussa. Näiden laajennusten myötä MoGo:sta tuli ensimmäinen go-tekoäly, joka saavutti go:ssa mestaritason *dan*-arvon (Gelly ja Silver, 2011). MCTS alkoi vakiinnuttaa asemaansa go-pelin tekoälyratkaisuissa, mutta todellinen läpimurto tapahtui kuitenkin vasta myöhemmin, AlphaGo:n myötä.

Go-tekoälyn ja siihen sovelletun MCTS:n voittokulun voidaan katsoa saavuttaneen tähänastisen kulminaatiopisteensä *DeepMind:in AlphaGo*:ssa. Kuten kohdassa 3.1.3 tarkemmin kuvattu ja myöhemmin julkaistu jälkeläisensä AlphaZero, AlphaGo käytti toiminnassaan neuroverkkojen ja MCTS:n yhdistelmää (Silver et al., 2016). AlphaGo kykeni hallitsevasti parempaan suoritukseen kuin muut aikaisensa huipputason go-tekoälyt, jotka sen julkistamisen aikaan olivat jo lähes poikkeuksetta myös MCTS-pohjaisia, voittaen ne 99.8% prosentissa peleistään (Silver et al., 2016). Laajempaan maailmanmaineeseen AlphaGo nousi voitettuaan sekä Euroopan go-mestarin Fan Huin että hieman myöhemmin Lee Sedolin, yhden vahvimmista pelaajista pelin historiassa (Silver et al., 2017). Oli selvää, että uusi merkittävä rajapyykki aiemmin huipputason ihmispelaajia heikommissa go-tekoälyn toteutuksissa oli saavutettu, ja että MCTS-puuhaku oli kehityksessä avainroolissa.

AlphaGo:n tuoreempi ja kehittyneempi versio, *AlphaGo Zero*, toimii go:n osalta pääpiirteittäin jotakuinkin samalla tavalla kuin kohdassa 3.1.3 kuvatussa shakin tapauksessa, pienillä toteutuseroilla. Toisin kuin shakissa, go:n peliasemat ovat *epävariantteja* suhteessa *käännöksiin* (rotation) tai *heijastuksiin* (reflection). Tätä hyväksikäytettiin AlphaGo Zero:n MCTS-vaiheen aikana siten, että peliasemiin sovellettiin satunnaisesti valittuja käännöksiä ja heijastuksia, jolloin Monte Carlo -evaluaatio tuli keskimääräisesti tasapainotetuksi vinoumia vastaan. (Silver et al., 2018)

Go-pelin kohdalla nähtiin, että perinteiset minimax- ja evaluaatiopohjaiset ratkaisut eivät kyenneet saavuttamaan korkeinta ihmispelaajatasoa. Pelin hakupuun syvyyttä raskaasti rajoittava korkea haarautumiskerroin ja etenkin korkean tason ihmispelaajien tietämyksen formalisoinnin ongelma aiheuttivat tällaisten ratkaisuiden kohdalla merkittäviä ongelmia. Vasta MCTS-tekniikoiden soveltaminen sysäsi go-tekoälyn valtavin loikkauksin uudelle aikakaudelle, kulminoituen AlphaGo:n hallitsevaan suoritukseen.

4.3 Pokeri

Tässä alaluvussa kuvataan lyhyesti pokerin Texas Hold'em -version säännöt ja rakenne, sekä pohditaan pokerin tekoälyn haasteita. Tämän jälkeen tutustutaan lyhyesti muutamiin aiemmin pelissä hyödynnettyihin tekoälymenetelmiin, ja lopuksi tarkastellaan vastustajien heikkouksia hyväksikäyttäviä MCTS-pohjaisia tekoälyratkaisuja.

4.3.1 Pokerin kuvaus – Texas Hold'em

Tässä luvussa pokerin ja sen tekoälyhaasteiden ja -ratkaisuiden tarkastelu keskittyy Texas Hold'em -pelimuotoon. Texas Hold'em peli koostuu neljästä pelivaiheesta, jotka ovat nimeltään *preflop*, *flop*, *turn* ja *river*. Preflop-vaiheessa pelin panokseen eli pottiin asetetaan kaksi pakollista panostusta, *pieni sokkopanos* (small blind) jonka asettaa jakajasta vasemmalla oleva pelaaja, ja *suuri sokkopanos* (big blind) jonka asettaa hänestä seuraava pelaaja. Suuri sokkopanos on yleensä erikseen sovitun minimipanostuksen suuruinen, ja pieni sokkopanos puolet tästä. Tämän jälkeen jokaiselle pelaajalle jaetaan kaksi käsikorttia, jotka pidetään piilossa muilta pelaajilta.

Pelaajien katsottua käsikorttinsa suuren sokkopanoksen asettajasta seuraava pelaaja aloittaa *panostuskierroksen*. Tällöin hänen on valittava kolmesta toiminnosta: Luovuttaminen eli ”kippaus” (fold), vaaditun panoksen asettaminen (check), tai korottaminen (raise). Luovuttaessaan pelaaja ei enää aseta pottiin lisäpanoksia, vaan luopuu pelistä ja osuudestaan potin sisältöön. Mikäli hän ”katsoo” eli asettaa vaaditun panoksen, kierros jatkuu seuraavaan pelaajaan. Korottaessaan pelaaja asettaa vaaditun panoksen lisäksi pottiin myös oman korotuksensa, joka nostaa vaadittua panosta muille pelaajille. Kierros jatkuu, kunnes kukin pelaaja on joko asettanut vaaditun panoksen tai luovuttanut.

Seuraavaksi pelataan flop, turn ja river -vaiheet, joissa pöytään asetetaan julkisia kortteja. Flop-vaiheessa niitä asetetaan kolme, ja turn- sekä river-vaiheissa yksi. Kunkin vaiheen lopussa suoritetaan jälleen panostuskierros edellä kuvatun kaltaisesti. Pelaajien käsien arvo on paras viiden kortin pokerikäsi, joka voidaan muodostaa pelaajan kätketyistä käsikorteista ja julkisista korteista. River-vaiheen jälkeen pelissä mukana olevat pelaajat paljastavat käsikorttinsa ja parhaan käden muodostava pelaaja voittaa potin. Tasatilanteessa potti jaetaan voittajien kesken.

4.3.2 Tekoälyn haasteet pokerissa

Billings et al. (2002) pohtivat pokerille ominaisia haasteita pelin tekoälytoimijoille laatiessaan omaa pokeritekoälyään *Poki*a. Pokeri pitää sisällään monia sellaisia piirteitä, jotka erottavat sen muista

peleistä kuten aiemmin tarkastelluista shakista ja go:sta, ja tekevät peliä korkealla tasolla pelaavan tekoälytoimijan laatimisen erittäin haastavaksi ja kiinnostavaksi. Pokeri on epätäydellisen tiedon peli, sillä pelin koko tila ei ole jatkuvasti kaikkien pelaajien tiedossa. Pelikorttien satunnaisuus tuo peliin myös vahvan stokastisen elementin. Toisin kuin shakki ja go, on tässä tutkielmassa käsitelty Texas Hold'em -pokerimuunnelma myös useamman kuin kahden pelaajan peli. Näistä syistä pelin pelaaminen korkealla tasolla vaatii kehittynyttä *riskienhallintakykyä*, kykyä vastustajien *harhaan johtamiseen*, sekä *vastustajien mallinnusta*.

Billings et al. (2002) listaavat ominaisuuksia, joita heidän mukaansa vaaditaan pokeria maailmanluokan tasolla pelaavalta pokerialgoritmilta. Ohjelman on kyettävä arvioimaan kätensä vahvuus muihin pelaajien nähden, perustuen omiin ja julkisiin kortteihin. Lisäksi sen on myös arvioitava käden potentiaali tulevaisuudessa, vielä jakamattomien julkisten korttien jälkeisessä pelitilassa. Pokerissa myös kyky *bluffata*, eli pyrkiä voittamaan heikolla kädellä, on erittäin tärkeä taito onnistuneelle pelaajalle. *Ennalta-arvaamattomuus* omassa pelissä on myös olennaista, sillä se vaikeuttaa vastustajien yrityksiä *mallintaa pelaajan strategiaa* tarkasti – taito, joka myös hyvin tärkeä menestyksekkäälle pokerin pelaajalle.

Pokerissa pelaajan mahdollisten toimintojen joukko on hyvin suppea, rajoittuen vain panostuskierroksien kolmeen mahdolliseen toimintoon, korotuksen tapauksessa sisältäen myös korotuksen suuruuden valinnan. Edellä mainituista ominaisuusvaatimuksista johtuen pätevän pokeritekoälyn on kuitenkin sisällytettäviä monia elementtejä omaan panostusstrategiaansa.

4.3.3 Pokeriin sovellettuja tekoälymenetelmiä

Rubin ja Watson (2011) tarkastelevat pokeriin sovellettuja erilaisia tekoälymenetelmiä olemassaolevaan kirjallisuuteen perustuen. *Tetämyspohjaiset järjestelmät* (knowledge-based systems), jotka pohjautuvat vahvasti asiantuntijapohjaiseen aluetietämykseen, jakautuvat tyypillisesti kahteen kategoriaan: *sääntöpohjaisiin asiantuntijajärjestelmiin* (rule-based expert systems) ja yleisluontoisempiin *kaavapohjaisiin menetelmiin* (formula-based methods).

Yksinkertaiset sääntöpohjaiset asiantuntijajärjestelmät perustuvat joukolle staattisia ennalta rakennettuja ehtolauseita koskien tilanteita, joita pelissä luultavammin tulee eteen. Kaavapohjainen menetelmä on hieman samankaltainen, mutta yleisluontoisempi. Menetelmässä jollekin kaavalle otetaan syötteeksi joukko nykyistä pelitilannetta kuvaavia tietueita, ja kaavan tuottamien arvojen perusteella valitaan satunnaisesti seuraava pelitoiminto. Useimmiten kaavalle syötteeksi tulevat

arvot ovat numeraalisia representaatioita *käden vahvuudesta* tai *pottitodenäköisyyksistä* (pot odds). Eräs yksinkertainen tapa määritellä käden vahvuus on suhteuttaa pelaajan käsikorteista ja julkisista korteista muodostettava vahvin käsi kaikkiin mahdollisiin käsiin, joita satunnaisella vastustajalla voi tilanteessa olla. Pottitodennäköisyydellä tarkoitetaan numeraalista määrittelyä sille, kuinka suotavaa pelaajan on jatkaa kierrosta, ottaen huomioon tehdyt panostukset ja mahdolliset tulevat palkkiot. (Rubin ja Watson, 2011)

Verrattain yksinkertaiset tietämyspohjaiset tekoälypelaajat eivät kuitenkaan kyenneet kovinkaan onnistuneesti haastamaan pätevämpiä ihmispelaajia tai kehittyneempiä tekoälytoimijoita. Eräs tällaisten tekoälyratkaisuiden suurimmista ongelmista on se, kuinka vaikeaa asiantuntija-aluetietämyksen formalisointi ehtolause- tai kaavapohjaiseen ohjelmalliseen muotoon voi olla. Järjestelmän kehittyessä ja kasvaessa siitä voi myös tulla hankalasti ylläpidettävä. Staattiseen asiantuntijatietämykseen pohjautuva tekoäly tuottaa myöskin todennäköisesti kaavamaista peliä pelaavan toimijan, jota vastustajat voivat helposti ennakoida ja hyväksikäyttää. Voidaankin todeta, että Texas Hold'em pitää sisällään liian laajan joukon mahdollisia tilanteita, jotta tietämyspohjainen tekoälytoimija voisi kyetä käsittelemään ne kaikki suotuisalla tavalla. (Rubin ja Watson, 2011)

4.3.4 MCTS:n soveltaminen pokerin tekoälyyn

Schweizerin et al. (2009) mukaan tehokkaalle pokeritekoälylle ei riitä, että se kykenee noudattamaan peliteoreettisessa mielessä *optimaalista* strategiaa, vaan kuten aiemmassakin osiossa todettiin, on se myös kyettävä havaitsemaan ja hyväksikäyttämään vastustajiensa heikkouksia. Yksinkertaisena peliteoreettisena esimerkkinä optimaalisesta strategiasta ja sen potentiaalisesta heikkoudesta toimii tunnettu kivi-paperi-sakset peli (lyh. KPS). Tässä pelissä jokaisen pelaajan optimaalinen strategia on valita satunnaisesti jokin kolmesta siirrosta, eikä tästä poikkeamalla voi kukaan yksittäinen pelaaja saavuttaa etua muihin tätä strategiaa noudattaviin pelaajiin nähden. Tällaista tilannetta kutsutaan *Nashin tasapainotilanteeksi* (Nash equilibrium). Jos jokin pelaaja kuitenkin poikkeaa optimaalisesta strategiasta, on vastustajan kannalta suotuisaa havaita tämä poikkeama ja mukauttaa strategiaansa sen vastaiseksi, siinä missä optimistrategian noudattaminen ei parantaisi voittomahdollisuuksia (Schweizer et al., 2009). KPS-pelin tapauksessa tilanne on yksinkertainen ja helposti havainnollistettavissa, mutta sama mekanismi pätee vahvasti myös pokerissa, jossa vastustajien mallinnus heidän heikkouksiensa hyödyntämiseksi on tärkeää.

Monte Carlo -puuhaun vahvuus pokerissa on Schweizerin et al. (2009) mukaan se, että se kykenee käsittelemään monia pelin osa-alueita ilman tarvetta eksplisiittiselle pelitietämyksen

formalisoinnille, joka olisi tarpeen silloin jos pelitilapuun solmuja jouduttaisiin arvioimaan evaluaatiofunktiolla syvyysrajoitetussa leveyshaussa. Aiemmin mainitut olennaiset pokeritekoälyn komponentit, kuten käden vahvuus ja potentiaali, panostusstrategia, bluffaaminen, ennalta-arvaamattomuus ja vastustajien mallinnus, tulevat kaikki implisiittisesti huomioon otetuksi MCTS-simulaatioita suorittamalla saadun tilastollisen datan kautta.

Schweizer et al. (2009) kehittivät *MCTS-pohjaisen pokeritekoälyn, AKI-RealBotin*. AKI-RealBot suorittaa kussakin valintatilanteessa kaksi itsenäistä MCTS-simulaatiosarjaa, yhden vaaditun panoksen asettamisen toiminnolle ja yhden korottamistoiminnalle. Luovuttamistoiminnon simulointi ei luonnollisestikaan ole tarpeen, sillä se päättää pelin pelaajan osalta. Simulaatioita suoritetaan rajoitetun ajan, ja sen tuottamia *odotusarvoja* (expected values) käytetään toiminnon valinnan arviointiin. AKI-RealBot kerää myös tilastollista dataa jokaisen vastustajan todennäköisyydestä luovuttaa, panostaa tai korottaa, ja suorittaa näin karkeaa vastustajan mallinnusta. Tekoälyn MCTS-simulaatioita suorittaessa vastustajan tuntemattomia käsikortteja ei valita puhtaan satunnaisesti, vaan valintaa painotetaan näiden vastustajamallien perusteella.

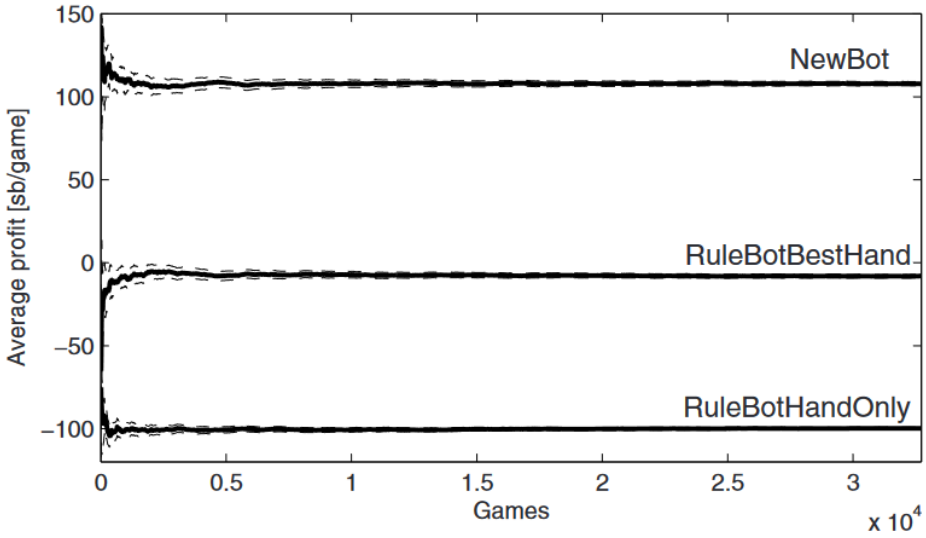
MCTS-puuhaun suorittaman odotusarvojen arvioinnin jälkeen AKI-RealBot suorittaa odotusarvoille jälkiprosessointia, jolla se pyrkii hyödyntämään heikkojen pelaajien heikkouksia. Heikot pelaajat määritellään tässä tapauksessa yksinkertaisen pelitietämyksen mukaan pelaajiksi, jotka luovuttavat liian usein ja helposti, tai toisen ääripään pelaajiksi, jotka pelaavat heikkoja käsiä liian pitkälle tai loppuun asti. MCTS-puuhaun tuottamista odotusarvoista ei valita suoraan parhaan arvon saanutta toimintoa, vaan odotusarvoihin sovelletaan *päätösrajoja* (decision bounds) heikkojen vastustajien kohdalla. Tällä metodilla AKI-RealBot painottaa toimintaansa siten, että se pyrkii pelaamaan mahdollisimman paljon käsiä heikkoja vastustajia vastaan (Schweizer et al., 2009).

AKI-RealBot osallistui *AAAI-08 Pokeritekoälykilpailuun*, jossa se sijoittui toiseksi. Taulukosta 1 nähdään, että korkea sijoitus johtui tekoälyn kyvystä hyödyntää kilpailun heikointa osallistujaa, GUS6-tekoälyä, jonka se ”lypsi kuiviin”. AKI-RealBot ei pärjännyt erityisen hyvin vahvemmille tekoälyille, joten ilman GUS6:n osallistumista sijoitus olisi jäänyt huomattavasti heikommaksi (Schweizer et al., 2009).

	POKI0	AKI-REAL	DCU	CMURING	MCBOT	GUS6
POKI0		65,176	2,655	18,687	29,267	214,840
AKI-REALBOT	-65,176		-15,068	-2,769	30,243	348,925
DCU	-2,665	15,068		7,250	16,465	90,485
CMURING	-18,687	2,769	-7,250		7,549	92,453
MCBOTULTRA	-29,267	-30,243	-16,465	-7,549		16,067
GUS6	-214,840	-348,925	-90,485	-92,453	-16,067	
Total	330,822	296,293	126,657	76,848	-67,529	-763,091
avg. winnings/game	3934	3579	1512	939	-800	-9042
SB/Hand	0.656	0.588	0.251	0.152	-0.134	-1.514
Place	1.	2.	3.	4.	5.	6.

Taulukko 1: AAAI-08 pokerikilpailun tulokset sijoituksen mukaan ja pareittain (Schweizer et al., 2009)

Myös Van den Broeck et al. (2009) kehittivät MCTS-pohjaisen pokeritekoälyn, joka edellä kuvatun Schweizerin et al. (2009) AKI-RealBotin tavoin pyrkii hyödyntämään vastustajien heikkouksia. He käyttivät toteutuksessaan luvussa 2 kuvattua MCTS-UCT -muunnelmaa, vastustajan mallinnusta sekä kehittämiään takaisinpropagaatio- ja valintastrategioita, jotka mallintavat eksplisiittisesti myös simulaationäytteiden epävarmuutta. Tällöin UCT-valinnan painotuskaavassa otetaan huomioon myös epävarmuustekijä, joka perustuu lapsisolmusta suoritetuille simulaatioille. Testatakseen tekoälyään he pelauttivat sitä sekä alkeellista että kehittyneempää sääntöpohjaista pelaajaa vastaan. Näissä kokeiluissa heidän tekoälynsä, nimeltään *NewBot*, oli selvästi sääntöpohjaisia pelaajia vahvempi (kuva 6). He pelauttivat NewBot:ia myös vakiomuotoista takaisinpropagaatiota käyttävää MCTS-tekoälyä vastaan, ja tässäkin kokeessa NewBot osoittautui vahvemmaksi. Lisäksi laskenta-ajan lisääminen merkitsi NewBot:in edun kasvamista entisestään (Van den Broeck et al., 2009).



Kuva 6: NewBot:in keskimääräiset voitot sääntöpohjaisia tekoälyjä vastaan (Van den Broeck et al., 2009)

Heinrich ja Silver (2015) kehittivät *pehmeää UCT:ta* (smooth UCT) käyttävän MCTS-UCT -tekoälytoimijan pokeriin. He tutkivat, voidaanko vakiomuotoisen UCT:n kyvyttömyys konvergoitua aiemmin kuvattuun Nashin tasapainotilaan ylittää siten, että kuitenkin säilytetään UCT:n nopea oppimistahti. Tähän tarkoitukseen he kehittivät pehmeän UCT-algoritmin, joka yhdistää *kuvitteellisen pelin* (fictitious play) käsitteen Monte Carlo -puuhakuun. Kuvitteellisella pelaajalla tarkoitetaan tässä pelaajaa, joka pyrkii aina suorittamaan parhaan mahdollisen toiminnon vastauksena muiden pelaajien keskimääräiseen pelistrategiaan. Tällainen strategia konvergoituu Nashin tasapainotilaan joissakin peleissä, kuten nollasummapeleissä ja potentiaalipeleissä. Heidän pokeritekoälynsä yhdistää tällaisen keskimääräisen strategian mukaan pelaamisen Monte Carlo -puuhaulle ominaiseen hyödyn maksimointiin pyrkivään toiminnanvalintastrategiaan (Heinrich ja Silver, 2015).

Kahdessa kevyemmässä ja Texas Hold'emiä rajoitetummassa pokerivariantissa, *Kuhnin pokerissa* ja *Leduc Hold'emissä*, pehmeä UCT kykeni oppimaan strategioita yhtä nopeasti kuin vakiomuotoinen UCT, mutta toisin kuin UCT, pehmeä UCT lähestyi Nashin tasapainotilaa. Rajoitetun panostuksen Texas Hold'emissä pehmeä UCT suoriutui vakiomuotoista paremmin ja voitti kolme hopeamitalia vuoden 2014 tietokonepokerikilpailussa (Annual Computer Poker Competition, ACPC). Näiden tulosten perusteella vaikuttaa siltä, että pehmeä UCT osoittaa lupaavaa kehitystä MCTS-algoritmien soveltamisessa epätäydellisen tiedon peleihin (Heinrich ja Silver, 2015).

Pokerin kohdalla nähtiin, että sen stokastinen ja ei-deterministinen luonne aiheuttavat tekoälylle monia haasteita. Monte Carlo -pohjaisten ratkaisuiden voidaan pokerinkin kohdalla katsoa soveltuvan perinteisiä puuhakualgoritmeja paremmin pelin laajojen ja syvien puurakenteiden tutkimiseen. MCTS-ratkaisut näyttävät suoriutuvan hyvin myös heikompitasoisten pelaajien heikkouksien hyväksikäytössä.

4.4 Magic: The Gathering (MTG)

Tässä alaluvussa kuvataan lyhyesti MTG-keräilykorttipelin perusrakenne ja pelin tekoälylle asettamat moninaiset haasteet. Tämän jälkeen tutustutaan kahteen pakanrakennukseen liittyvään tekoälyratkaisuun tai ratkaisujoukkoon, ja lopuksi tutustutaan MCTS-algoritmiin pohjautuviin MTG-tekoälyratkaisuihin.

4.4.1 MTG:n kuvaus

Magic: The Gathering (MTG) on vuonna 1993 julkaistu keräilykorttipeli, jota voidaan perustellusti pitää muiden nykyään aktiivisesti pelattujen kilpailullisten keräilykorttipelien edeltäjänä. MTG:tä pelataan edelleen muiden uudempien tulokkaiden rinnalla maailmanlaajuisesti ja erittäin kilpailullisesti sekä perinteisesti fyysisillä korteilla että lisääntyvissä määrin digitaalisessa muodossa (MTG Online ja MTG Arena). MTG:n voidaan katsoa olevan tässä tutkielmassa tarkemmin tarkastelluista peleistä säännöiltään, rakenteeltaan ja sisällöltään selkeästi monimutkaisin, ja näitä piirteitä avataan tässä kohdassa pääpiirteittäin, mutta ei täysin kattavasti. Pelin täydet säännöt ovat luettavissa Magic: the Gatheringin kattavasta sääntökirjasta (Wizards of the Coast, 2023).

Perusrakenteeltaan MTG on kahden pelaajan peli, jossa kumpikin pelaaja rakentaa korteista itselleen pakan, joissa on peliformaatista riippuen vaihteleva minimimäärä kortteja. Kahdessa pääformaattityypissä pakan minimikoko on *limited*-formaatissa 40 korttia ja *constructed*-formaatissa 60 korttia. Kortteja saa olla minimimäärää enemmänkin, mutta yleensä tämä ei ole pakan optimoinnin kannalta suositeltavaa. Pelin päävoittokeino on laskea vastustajan *elämäpisteet* aloitustilanteen kahdestakymmenestä nollaan, mutta pelissä on mahdollista saavuttaa myös muita, harvinaisempia voittokeinoja. (Wizards of the Coast, 2023)

Pelin aloittaja arvotaan, ja pelin alussa kumpikin pelaaja nostaa sekoitetusta pakastaan käteensä seitsemän korttia, ja ensimmäinen pelaaja aloittaa vuoronsa. Vuoro rakentuu viidestä eri vaiheesta: *Aloitusvaihe*, *ensimmäinen päävaihe*, *taisteluvaihe*, *toinen päävaihe*, ja *lopetusvaihe*. Vuoronsa

päävaiheissa pelaajat voivat pelata kädestään kortteja, jotka joko jäävät pelikentälle (*permanentit*) tai aiheuttavat jonkin vaikutuksen ja siirtyvät sen jälkeen pelaajan *hautausmaalle* (graveyard), josta niitä ei voi ilman erillisiä keinoja enää pelata. Tällaisia ei-pysyviä kortteja ovat *sorceryt*, joita voi pelata ainoastaan oman vuoronsa päävaiheissa, sekä *instantit*, joita voi pelata milloin tahansa, myös vastustajan vuorolla. Jokaisen vuoron alussa, pois lukien aloittavan pelaajan ensimmäinen vuoro, nostaa vuorossa oleva pelaaja pakastaan yhden kortin. Pelattavien korttien määrälle kussakin vuorossa ei ole pelin perussääntöjen puolesta ylärajaa, kunhan niiden hinnan pystyy maksamaan. (Wizards of the Coast, 2023)

Korteilla on lähes aina myös hinta, joka maksetaan maakorteilla, jotka ovat pysyviä kortteja. Myös muilla pysyväiskorteilla saattaa olla aktivoitavia ominaisuuksia, joiden käytöllä on hinta. Pelaaja voi vuorollaan pelata normaalisti vain yhden maakortin, ja kutakin pelattua maakorttia voi kerran vuorossa käyttää tuottamaan yhden *manan*, joilla muiden korttien tai aktivoitavien ominaisuuksien hinta maksetaan. Käytön jälkeen maakortti pysyy pelissä ja on jälleen käytettävissä pelaajan seuraavalla vuorolla. Pelaajan maksukyky siis yleensä nousee pelin edetessä, mikäli hän pelaa pöytään jatkuvasti lisää maakortteja. MTG:ssä useimmilla korteilla on yksi tai useampi *väri*, jolloin osa kortin manahinnasta täytyy maksaa kyseisellä värillä. Värejä on viisi: sininen, valkoinen, vihreä, punainen ja musta. Myös vakiomuotoisia maakortteja on viittä eri tyyppiä, joista kukin tuottaa tietyn väristä manaa. (Wizards of the Coast, 2023)

Pysyväiskorteista olennaisimpia ovat yleensä oliokortit, joilla voi taisteluvaiheessa hyökätä vastustajaan. Vastustaja voi puolestaan torjua olioita omilla olioillaan, jolloin oliot taistelevat omien arvojensa (*voima* ja *kestävyys*) mukaisesti keskenään, jolloin yksi tai useampi olio saattaa tuhoutua ja siirtyä omistajansa hautausmaalle. Mikäli oliota ei torjuta, se vähentää vastustajan elämäpisteitä oman voimansa verran. (Wizards of the Coast, 2023)

MTG:n monimutkaisuutta lisää se, että vaikka pelin erilaisissa kilpailullisissa formaateissa onkin sallittu vain tietty korttijoukko, on niitä silti useimmissa pelimuodoissa käytössä merkittävä määrä. Yhteensä MTG:ssä on tämän kirjoittamisen hetkellä yli 25 000 erilaista korttia, ja lisää julkaistaan vuosittain. Lisäksi korttien vaikutukset ja ominaisuudet voivat täysin vapaasti vuorovaikuttaa keskenään ja vaikuttaa pelin etenemiseen mielivaltaisin tavoin, ja jopa muokata tai kumota aiemmin kuvattuja pelin perussääntöjä.

4.4.2 Tekoälyn haasteet MTG:ssä

MTG on pokerin tavoin stokastinen epätäydellisen informaation peli. Kuten pokerissa, pelaaja ei näe vastustajansa käsikortteja, eikä tiedä mitä kortteja tulee nostamaan omaan käteensä. Lisäksi, toisin kuin pokerissa, on joissakin tilanteissa myös vastustajan pakan sisältö pelaajalle lähes täysin tuntematon - se voi sisältää mitä tahansa kyseisessä formaatissa sallittuja kortteja, joita voi olla jopa tuhansia.

Edellä mainituista syistä vahvan tekoälypelaajan toteuttaminen MTG:hen on haasteiltaan hyvin lähellä yleisen tekoälytoimijan luomista, sillä erilaisilla pakoilla pelatut pelit voivat olla korttien sääntöihin vaikuttavan luonteen vuoksi kuin täysin erillisiä pelejä. Tästä syystä Monte Carlo -puuhaun yleispätevän luonteen voidaan alustavasti ajatella soveltuvan erityisen hyvin MTG:n tekoälyratkaisuiden kehittämiseen. (Ward ja Cowling, 2009)

4.4.3 MTG:hen sovellettuja tekoälymenetelmiä

Toistaiseksi MTG:hen ei ole sen haastavasta luonteesta johtuen kehitetty vahvaa tekoälyä (Ward et al., 2021). Uusin MTG:n digitaalinen muoto, *MTG Arena* -sovellus, sisältää alkeellisen tekoälyn, joka pelaa vain verrattain pientä joukkoa kohtalaisen heikkoja pakkoja ja kortteja, eikä kykene mielekkäällä tavalla haastamaan hiemankaan edistyneempää pelaajaa. Kyseinen S*parky*-tekoäly soveltuukin korkeintaan hyvin kevyeen pakan testaamiseen.

MTG eroaa pokerin kaltaisista staattisen pakan pelikorttipeleistä myös sikäli, että pakanrakennuksen voidaan katsoa olevan erittäin tärkeä osa MTG:n pelaamista. Hyvin rakennettu pakka antaa yleisellä tasolla merkittävän edun heikompaa pakkaa vastaan pelatessa, jopa siinä määrin, että se riittää kumoamaan suurenkin pelitaitoeron vahvemman pakan pelaajan eduksi. Pakanrakennuksen rajoitteet riippuvat pelattavasta pääformaatista - limited-formaateissa pakkaan valittavana on vain pieni satunnainen joukko kortteja, kun taas constructed-formaateissa saatavilla on kaikki kyseisessä formaatissa lailliseksi määritellyt kortit, eli käytännössä aina huomattavasti suurempi määrä. Tällöin kyseessä on *kombinatorinen optimointi-ongelma* erittäin suuressa hakuavaruudessa (Bjørke ja Fludal, 2017).

Bjørke ja Fludal (2017) kehittivät ratkaisua *MTG:n pakanrakennuksen kortinvalintaongelmaan geneettistä algoritmia* hyödyntäen. Geneettistä algoritmia sovellettaessa on alussa olemassa johonkin ongelmaan ratkaisuehdotusten joukko, jota kutsutaan populaatioksi. Näihin ratkaisuehdotuksiin kohdistetaan mutaatioita, jolloin ne muuttuvat - joistakin tulee soveltuvampia,

toisista heikompia. Joillakin määritellyillä ehdoilla heikot ratkaisut poistetaan. Tätä prosessia toistetaan, kunnes saavutaan tyydyttävään tulokseen tai saavutetaan jokin määritetty toistojen yläraja. Vaiheittain kuvattuna geneettinen algoritmi etenee seuraavasti (Bjørke ja Fludal, 2017) :

1. Alustus: Luodaan alkupopulaatio generoimalla satunnaisratkaisuja, joko tasaisen satunnaisesti koko ratkaisuavaruuden alueella, tai jotakin tiettyä jo hyväksi epäiltyä aluetta painottaen.

2. Valinta: Osa populaatiosta valitaan tuottamaan seuraava sukupolvi. Yleensä paremman sopivuusarvon ratkaisuilla on korkeampi todennäköisyys tulla valituksi, mutta valintamenetelmät voivat vaihdella.

3. Lisääntyminen: Tuotetaan uusi sukupolvi valintavaiheessa valituista ratkaisuista. Nämä jälkeläiset muodostetaan yhdistelemällä jollakin tavalla, esimerkiksi satunnaisesti, kahden vanhemman ominaisuuksia. Tässä vaiheessa voidaan myös mutatoida jälkeläisiä jollakin halutulla tavalla. Näin saadaan lopulta uusi populaatio.

Edellä kuvattua menetelmää toistetaan, kunnes saavutetaan jokin määritelty lopetuskriteeri. Bjørke ja Fludal (2017) kehittivät geneettistä algoritmia hyödyntävän pakanrakennusjärjestelmän, joka kykeni luomaan laadukkaita pakkoja, jotka vertautuivat hyvin kokeneiden ihmispelaajien luomiin pakkoihin. Nämä pakat kykenivät ylläpitämään yli 50% voittoprosenttia kuutta hyvin erilaista vastustajaa vastaan. Pakat eivät kuitenkaan heidän mukaansa sisältäneet korkean tason ihmispelaajien luomille pakoille ominaisia piirteitä.

Ward et al. (2021) tutkivat *pakanrakennusongelmaa draft-peliformaatin* kontekstissa. Draft on aiemmin kuvatun limited-formaatin kahdeksan pelaajan pelimuoto, jossa jokaiselle pelaajalle jaetaan yksi viidentoista osittain satunnaisen kortin pakkaus. Kukin pelaaja valitsee pakkauksesta itselleen yhden kortin, ja antaa sitten loput viereiselle pelaajalle. Tätä toistetaan, kunnes kaikki kortit on valittu. Prosessi toistetaan vielä kaksi kertaa, ja tämän jälkeen kukin pelaaja rakentaa itselleen pakan valitsemistaan korteista. Pakan koolla ei ole ylärajaa, mutta yleensä pakkaan suositellaan sisällytettäväksi 20-25 korttia, joista yhdessä vapaassa käytössä olevien maakorttien kanssa voi muodostaa 40 kortin pakan. Valitut kortit ja pakkojen sisältö eivät ole muille pelaajille julkista tietoa. Pakanrakennuksen jälkeen pelaajat pelaavat pakoillaan turnauksen millä tahansa valitulla turnausmuodolla.

Kortinvalintavaihe sisältää monia pätevän tekoälytoteutuksen kannalta mielenkiintoisia haasteita.

Arvioidessaan korttipakkauksesta tehtävää kortin valintaa tekoälyn tulisi ottaa huomioon yksittäisten korttien vahvuuden lisäksi myös jokaisen kortin mahdolliset synergiat jo valittujen korttien välillä, sekä myös vastustajien mahdollisesti käyttämät strategiat. Vastustajien mahdollisia strategioita voi pyrkiä arvioimaan kiinnittämällä huomiota esimerkiksi siihen, mitkä kortit eivät yhden täyden kierroksen jälkeen ole tulleet kenenkään valitsemaksi. (Ward et al. 2021)

Ward et al. (2021) kehittivät viisi draft-toimijaa, joista jokainen käyttää erilaista strategiaa kortinvalinnassaan. Esimmäinen, *RandomBot*, valitsee kortit täysin satunnaisesti. *RaredraftBot* puolestaan toimii hieman kokemattomien ihmispelaajien tapaan, valiten harvinaisuusluokaltaan harvinaisimman kortin. MTG:ssä harvinaisuusluokkia on neljä: *mythic rare*, *rare*, *uncommon* ja *common*, harvinaisuusjärjestyksessä lueteltuna. Korttien mahdollinen rahallinen arvo sekä keskimääräinen voimataso seuraavat harvinaisuusluokitusta jossakin määrin, joskin hyvin löyhästi ja runsain poikkeuksin. Kolmas toimija, *DraftsimBot*, käyttää monimutkaista heuristiikkaa kortinvalinnassaan. Heuristiikka perustuu sekä ihmisasiantuntijan määrittelemille korttivahvuuksille että kortin värin sopivuudelle jo valittujen vahvojen korttien kanssa. Neljäs toteutus, *BayesBot,* painottaa korttipareja, joita on harjoitusdrafteissa tilastollisesti usein valittu yhdessä. Viides tekoälytoimija, *NNetBot*, puolestaan pyrkii naiivia neuroverkkoratkaisua hyödyntämällä matkimaan ihmismäistä kortinvalintaa. (Ward et al. 2021)

Toimijoiden suorituksen ihmismäisyyttä arvioitiin vertaamalla jokaisen toteutuksen kykyä ennakoida oikeita ihmisten tekemiä valintoja, perustuen 21 590 oikean draftin testijoukkoon. Kuten olettaa saattaa, jokainen kehittyneempi toimija suoriutui tehtävästä paremmin kuin RandomBot ja RaredraftBot. NNetBot osoittautui kuitenkin joukon vahvimmaksi merkittävällä erolla. (Ward et al. 2021)

4.4.4 MCTS:n soveltaminen MTG:n tekoälyyn

Kuten aiemmin todettiin, vastaavat MTG:n tekoälyn toteuttamisen haasteet hyvin läheisesti yleisen pelitekoälytoimijan toteuttamisen haasteita. Mielekkään evaluaatiofunktion luominen olisi MTG:n tapauksessa erittäin hankalaa sen epätäydellisen informaation, stokastisen luonteen, ja ennen kaikkea korttien valtavan määrän ja niihin liittyvien sääntömuunnosten ja keskinäisten vuorovaikutusten vuoksi. Jopa pelitilanne, joka pintapuolisessa tarkastelussa näyttää olevan pelaajalle varma voitto, saattaa se kokeneen pelaajan intuition pohjalta olla kaikkea muuta, sillä yksikin kortti voi kääntää edun päälaelleen. Tästä yleisyysvaatimuksesta johtuen MCTS saattaakin tarjota ratkaisuja vahvan MTG-tekoälytoimijan luomiseen (Ward ja Cowling, 2009).

Ward ja Cowling (2009) tutkivat *Monte Carlo -puuhaun soveltamista MTG:n pelattavan kortin valintaan*. He vertasivat MCTS-algoritmiä käyttävää valinta-algoritmiaan kokeneen pelaajan kehittämään sääntöpohjaiseen tekoälyratkaisuun ja tutkivat, kykeneekö heikko (satunnainen) simulaatiopohjainen pelaaja voittamaan vahvan sääntöpohjaisen ratkaisun, ja mikäli näin on, millaisella simulaatiomäärällä. Kokeessaan he keskittyivät erityisesti olioiden väliseen taisteluun, joka on yksi MTG-pelin tärkeimmistä vuoron vaiheista. Useimmat MTG-pakat, etenkin aloittelijatasolla, rakentuvat vahvasti oliokorttien ympärille. Ward ja Cowling (2009) rajasivat testiympäristönsä kattamaan ainoastaan maa- ja oliokortteja, jotta pelin sääntöjoukko pysyisi yksinkertaisempana. Kokeessa pelitoimijoiden pakassa oli ainoastaan näitä korttityyppejä, ja ne ovat yksivärisiä. Pakat olivat myös kaikki identtisiä, jotta pakkojen mahdolliset vahvuuserot eivät vaikuttaisi lopputulokseen.

Edellä mainitulla tavalla rajatussa testiskenaariossa tekoälytoimija joutui tekemään kolmenlaisia päätöksiä: Millä oliolla hyökätä vastustajaan, miten torjua vastustajan oliot omilla olioilla silloin, kun häneen itseensä hyökätään, sekä päättää, mitä kortteja pelata kädestään omalla vuorollaan taisteluvaiheen ulkopuolella. Koska instant-tyyppiset missä tahansa tilanteessa pelattavat kortit rajattiin pois koejärjestelystä, kaksi ensimmäistä päätöstyyppiä ovat käytännössä täydellisen tiedon päätöksiä. Kolmas päätöstyyppi sen sijaan ei ole, sillä vastustajan käsikortit ovat tuntemattomia ja pelaajien tulevat kortinnostot satunnaisia. Nämä tekijät vaikuttavat kuitenkin vahvasti siihen, minkä kortin pelaaminen pöytään on milloinkin optimaalista. Etenkin keskipelissä, jolloin pelaajilla on yleensä eniten valinnanvaraa, haarautumiskerroin mahdollisten päätösten suhteen on korkeimmillaan. (Ward ja Cowling, 2009)

Jokaiselle päätöstyypille Ward ja Cowling (2009) toteuttivat sekä satunnaisen että kokeneen pelaajan aluetietämykseen nojaavan sääntöpohjaisen prosessin. Lisäksi he laativat korttien pelaamispäätöksiin UCB-algoritmia käyttävän MCTS-prosessin. Lopulta he loivat yhteensä 12 tekoälypelaajaa, joista kukin käytti jotakin yhdistelmää näistä kolmesta tekniikasta pelin eri vaiheessa, tosin siten, että MCTS-tekniikkaa sovellettiin ainoastaan pelattavan kortin valintaan. MCTS-toimijoille sallittiin 350 simulaatiota kortin valitsemiseen.

Ward ja Cowling (2009) pelauttivat erilaisia tekoälytoteutuksiaan siten, että kukin pelasi 100 peliä jokaista toista vastaan sekä aloittavana että toisena pelaajana. Satunnaistoteutusta käyttävät toimijat pärjäsivät näissä peleissä huonosti, kuten oletettavissa olikin. MCTS-toimijat sen sijaan kykenivät tehokkaasti haastamaan sääntöpohjaiset toteutukset, noin 5% keskimääräisellä edulla

voittoprosenteissa. Lisäksi Ward ja Cowling (2009) pelauttivat MCTS-kortinvalitsijatoimijaa sääntöpohjaista heuristiikkaa kaikissa päätöstyypeissään käyttävää tekoälyä vastaan, tarkoituksenaan testata simulaatiomäärän vaikutusta suorituskykyyn. He havaitsivat, että jo erittäin pienellä 10 simulaation määrällä MCTS-toimija saavuttaa huomattavasti satunnaistoimijaa parempia tuloksia, voittoprosenttien ollessa vastaavasti 35% ja 16%. Jo 200 simulaation kohdalla saavutettiin 50% voittoprosentti. Simulaatioiden määrän lisääminen tämän pisteen jälkeen ei kuitenkaan enää merkittävästi parantanut tuloksia. Tuloksista nähdään kuitenkin, että Monte Carlo -simulaatioiden avulla kyetään hyvin haastamaan satunnaispelaajaa huomattavasti vahvempi sääntöpohjainen pelaaja. (Ward ja Cowling, 2009)

Kuten edellä todettiin, sekä epätäydellinen tieto vastustajien korteista että kortinvedon satunnaisuus ovat MTG-tekoälyn suurimpia haasteita, sillä ne nostavat haarautumiskertoimen liian korkeaksi perinteisillä evaluaatiopohjaisilla puuhakutekniikoilla käsiteltäväksi. Cowling et al. (2012) tutkivat *determinisaatioiden (determinization) hyödyntämistä MTG:n MCTS-ratkaisuissa*. Determinisaatioissa kaikki kätketty ja satunnainen tieto oletetaan tunnetuksi. Tämä toteutetaan rakentamalla samasta juurisolmusta useita MCTS-puita, joista jokainen tutkii yhtä *mahdollista tulevaisuutta* kaikkien *mahdollisten tulevaisuuksien* joukosta. Kun kunkin tällaisen puun, eli *determinisaation*, simulaatioissa saavutaan pelitilaan, jossa vedetään tuntematon kortti, arvotaan se satunnaisesti ja "lukitaan" se puuhun. Kun samaan solmuun saavutaan myöhemmissä simulaatioissa, siirrytään suoraan seuraavaan lapsisolmuun. Puun kasvaessa luodaan tällä tavalla käytännössä yksi staattinen korttijärjestys kunkin pelaajan pakalle. (Cowling et al., 2012)

Koeasetelmaansa varten Cowling et al. (2012) toteuttivat seitsemän UCT-pohjaista MCTS-toimijaa. Jokaisen toimijan UCT-vakio, joka määrittää uusien puun haarojen tutkimisen painon simulaatioissa, asetettiin arvoon *1.5*. Samoin jokaisen toteutuksen puukohtaisten simulaatioiden määrä sai saman arvon, *250*. Toteutukset kuitenkin erosivat toisistaan sekä simulaatiostrategioidensa että puurakenteidensa suhteen. Lisäksi he toteuttivat perinteisiä vahvoja sääntöpohjaisia toimijoita, jotka toimivat kokeissa pohjaverrokkina. (Cowling et al., 2012)

Ensimmäisenä pelautettiin determinisaatioita käyttämätöntä, vakiomuotoisempaa MCTS-toteutusta sääntöpohjaisia tekoälyjä vastaan. Kyseinen toteutuksen puurakenne on luvussa 2 kuvatun expectiminimax-algoritmin puuhaun kaltainen siten, että jokaisessa satunnaisuutta sisältävässä solmussa luodaan mahdollisuussolmuja jokaisesta mahdollisesta tapahtumasta. Tällöin puun solmujen määrä kasvaa erittäin nopeasti, mikä todennäköisesti heikentää MCTS:n toimintaa.

Kyseinen toteutus suoriutuikin erittäin heikosti sääntöpohjaisia tekoälyjä vastaan, saavuttaen vahvimman sellaisen kohdalla vain 23% voittoprosentin. (Cowling et al., 2012)

Determinisaatioita hyödyntävä MCTS-toteutus suoriutui sääntöpohjaisia tekoälyjä vastaan huomattavasti vahvemmin. Toteutuksen käyttöön annettiin 10 000 simulaatiota, ja pelautuksessa testattiin myös tämän simulaatiobudjetin jakamista eri määrälle determinisoituja puita. Huomattiin, että keskimäärin parhaita tuloksia saavutettiin silloin, kun determinisaatioita oli 20 - 100 kappaletta siten, että kullakin oli käytettävissään 500 - 1 000 simulaatiota. Determinisaatiototeutuksella saavutettiin vahvinta sääntöpohjaista toimijaa vastaan parhaimmillaan 36% voittoprosentti, 20 determinisaation ja 500 simulaation budjettijaolla. (Cowling et al., 2012)

MTCS-toteutuksia keskenään pelautettaessa huomattiin, että kaikki determinisaatioita hyödyntävät toimijat suoriutuivat huomattavasti vakiomuotoisempaa ratkaisua vahvemmin. Näistä ratkaisuista kaksi paranneltua determinisaatiotoimijaa, *binääripuuratkaisu* sekä *mielenkiintoisten simulaatioiden ratkaisu*, nousivat hieman ylitse muiden. MTG:ssä pelaaja voi pelata useita kortteja kerralla, mutta binääripuuratkaisussa jokainen kortinvalintasolmu laajennetaan puuksi, jossa jokaisella solmulla on vain kaksi lapsisolmua - pelataanko jokin yksittäinen kortti, vai jätetäänkö se pelaamatta ja pelataan jokin muu kortti (tai ei korttia ollenkaan). Tällä tavalla yksi päätös jaetaan useamman, pienemmän päätöksen sarjaksi. Etuna on se, että päätöksen eri osat saavat simulaatioissa omat vahvuusarvonsa. Tällaisessa ratkaisussa on suotavaa, että tärkeimmät kortit sijaitsevat binääripuun yläosissa. (Cowling et al., 2012)

Mielenkiintoisten simulaatioiden tekniikassa puolestaan suositaan determinisaatioita, joissa korttien järjestys on "mielenkiintoinen". Mielenkiintoisuus määritellään sellaiseksi järjestykseksi, jossa toiminnan suorittamatta jättäminen (eli vuoronsa ohittaminen) antaa tietyn simulaatiomäärän suorittamisen jälkeen erilaisen tuloksen kuin sääntöpohjaisen tekoälyn ehdottaman toiminnon suorittaminen. Kahdesta tällaisesta toteutuksesta vahvemmaksi osoittautui versio, jossa 5% simulaatiobudjetista käytettiin mielenkiintoisten determinisaatioiden löytämiseen, toisen toteutuksen 1% budjettiin verrattuna. (Cowling et al., 2012)

Lopuksi Cowling et al. (2012) pelauttivat useamman parannellun determinisaatio-MCTS:n yhdistelmätoteutuksia sääntöpohjaisia toimijoita vastaan. Nämä toteutukset suoriutuivat huomattavasti vakiomuotoista determinisaatioratkaisua vahvemmin, saavuttaen 43.6% - 50.5% voittoprosentteja. Simulaatiobudjetin nosto 100 000:n lisäsi voittoprosentteja hieman.

Binääripuutoteutus kykeni tällä budjetilla saavuttamaan 52% voittoprosentin parasta sääntöpohjaista toimijaa vastaan, osoittaen MCTS-toimijoiden sopivilla parannuksilla ehostettuna kykenevän haastamaan vahvan sääntöpohjaisen tekoälyn, joka puolestaan oli aiemmin osoittautunut kilpailukykyiseksi vahvoja ihmispelaajiakin vastaan. (Cowling et al., 2012)

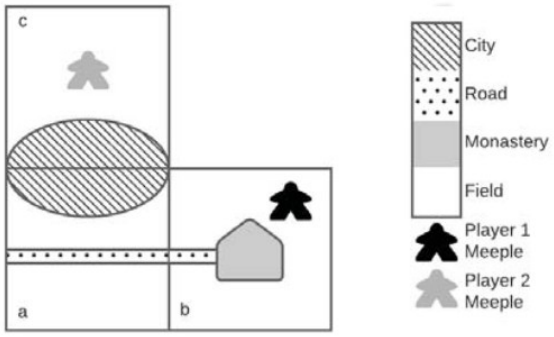
Molemmat tässä alaluvussa käsitellyt MCTS-lähestymistavat käyttivät koejärjestelyissään staattisia, ennalta määriteltyjä pakkoja, joiden sisältö oletetaan tunnetuksi. Tällaisissa tilanteissa MCTS-pohjaiset ratkaisut toimivat tarkastellun kirjallisuuden perusteella hyvin, haastaen sofistikoituneetkin sääntöpohjaiset toimijat. Monissa ammatti- ja amatööritason turnauksissa pelaajien pakat ovatkin julkista tietoa, joten tässä mielessä koeasetelma ei ole poikkeuksellinen. Monissa muissa huomattavasti yleisemmissä pelitilanteissa tilanne ei kuitenkaan ole koeasetelmien kaltainen. Suositussa MTG Arena -sovelluksessa valtaosa peleistä pelataan ennalta tuntematonta, järjestelmän määrittelemää vastustajaa vastaan. Tällöin myös vastustajan pakan sisältö on pelin alussa pelaajalle täysin tuntematon, joten täysipainoisen MTG-tekoälyn tulisi kyetä hyvään pelisuoritukseen myös tällaisessa tilanteessa. Vaikka tällaista tekoälytoimijaa ei nähtävästi olekaan vielä toteutettu, voidaan perustellusti uskoa, että MCTS-pohjaiset ratkaisut todennäköisesti ovat huomattavasti perinteisiä evaluaatiopohjaisia puuhakuratkaisuja lupaavampi kehityspolku myös tällä saralla.

5 Katsaus MCTS-algoritmin käyttöön muissa vuoropohjaisissa strategiapeleissä

Tässä luvussa tarkastellaan MCTS-algoritmin hyödyntämistä viidessä muussa vuoropohjaisessa strategiapelissä: *Carcassonnessa*, *Diplomacyssa*, *Settlers of Catan*issa, *Spades*-korttipelissä sekä *Hearthstone*-korttipelissä. Kunkin peliin kohdalla kuvataan pelin säännöt ja kulku lyhyesti, joskaan ei täysin kattavasti. Tarkastelun pääpaino on MCTS-algoritmin hyödyntämisellä ja sillä saavutetuilla tuloksilla. Toisin kuin edellisessä luvussa, tekoälyn haasteisiin tai aiempiin tekoälyratkaisuihin kunkin pelin kohdalla ei tarkemmin perehdytä.

5.1 Carcassonne

Ameneyro et al. (2020) tutkivat MCTS-algoritmin soveltuvuutta suosittuun Carcassonne-lautapeliin**.** Carcassonne on vuoropohjainen lautapeli, jossa pelaajat asettavat vuoron perään pelilaattoja sekä pelinappuloita, *meeplejä*, pelialueelle. Tavoitteena on kerätä mahdollisimman paljon pisteitä, ja pelin lopussa enemmän pisteitä kerännyt pelaaja voittaa. Peli alkaa siten, että pelialueella on ainoastaan yksi aloituslaatta. Vuorollaan kukin pelaaja nostaa sokkona yhden satunnaisen laatan laattavarastosta, ja asettaa sen pelialueelle jonkin alueella jo olevan laatan viereen siten, että laatat muodostavat eheän kokonaisuuden. Tämän jälkeen pelaaja saa halutessaan sijoittaa yhden seitsemästä meeplestään asettamalleen laatalle. Meepleä ei saa kuitenkaan asettaa niin, että se sijaitsisi sellaisella keskeneräisellä alueella, jossa on jo vastustajan meeple. Mikäli asetettu pelilaatta rakentaa loppuun jonkin pelin alueista, esim. kaupungin (kuten kuvassa 7), kyseinen alue pisteytetään ja pisteet lisätään sille pelaajalle, jolla oli kyseisellä alueella eniten meeplejä. Tällöin kaikki alueella sijainneet meeplet myös palautetaan takaisin pelaajiensa käyttövarantoon. Tasatilanteessa kumpikin pelaaja saa alueen pisteet. (Ameneyro et al., 2020)



Kuva 7: Laatan 'c' asettaminen rakensi kaupungin (city) valmiiksi (Ameneyro et al. 2020)

Carcassonne on täydellisen informaation peli, sillä koko pelitila, mukaan lukien kunkin pelaajan vuorollaan nostama laatta, on kaikkien pelaajien tiedossa. Laattojen nostaminen satunnaisesti tuo kuitenkin peliin stokastisen elementin. Pelikentän kasvaessa pelattujen laattojen myötä kasvaa myös pelaajien mahdollisten laatanasetuspaikkojen määrä. Lisäksi pelissä voi ”hyökätä” toisen pelaajan hallussa olevalle keskeneräiselle alueelle yhdistämällä tähän omalla laatallaan omassa hallussa olevan alueen, jolloin kullakin pelaajalla voi olla samalla keskeneräisellä alueella yksi tai useampi meeple.

Ameneyro et al. (2020) pelauttivat keskenään kolmea erilaista Carcassonne-tekoälyä. Yksi näistä oli vakiomuotoiseen MCTS-algoritmiin perustuva, toinen puolestaan hyödynsi MCTS:n RAVE-muunnelmaa, joka on kuvattu luvussa 2. Kolmas nojasi perinteisempään *Star2.5*-ratkaisuun, joka käytti expeciminimax-algoritmia alfa-beta-karsimisella vahvistettuna. Myös nämä minimax-laajennukset on kuvattu tarkemmin luvussa 2.

Ameneyron et al. (2020) pelikokeet osoittivat, että vakiomuotoinen MCTS-algoritmi kykeni kahden pelaajan Carcassonnessa huomattavasti Star2.5-algoritmia vahvempaan peliin, voittaen 88.57% peleistä ollessaan aloittavana pelaajana, ja 77.14% pelatessaan toisena. Star2.5-algoritmin tapauksessa vastaavat lukemat olivat 22.86% ja 2.86%. MTCS-RAVE -pohjainen ratkaisu suoriutui Star2.5:ttä vastaan aavistuksen heikommin, mutta silti hallitsevasti. Vakiomuotoisen MTCS-tekoälyn ja RAVE-muunnelman kamppailu oli huomattavasti tasaisempi, RAVE:n voittaessa 51% aloittamistaan ja 36% toisena pelaamistaan peleistä, siinä missä vakiomuotoisen MCTS:n vastaavat

lukemat olivat 64% ja 49%. Ameneyron et al. (2020) mukaan näyttäisikin siltä, että MCTS-pohjaiset ratkaisut heuristiikalla paranneltuna saattaisivat kyetä vahvempaan peliin kuin mikään Star2.5-pohjainen toimija.

5.2 Diplomacy

Theodoridis ja Chalkiadakis (2020) tutkivat kehittämiensä kahdeksan erilaisen MCTS-tekoälytoimijan suorituskykyä strategisessa Diplomacy-lautapelissä. Diplomacy on seitsemän pelaajan peli, jossa kukin pelaajista pelaa yhtä Euroopan ”suurvalloista” aikana ennen ensimmäistä maailmansotaa: Venäjää, Turkkia, Itävaltaa, Italiaa, Britanniaa, Saksaa tai Ranskaa. Kartta koostuu 75 provinssista, joista 34 on huoltokeskuksia (Supply Center, SC). Pelin voittaa pelaaja, joka ensimmäisenä valloittaa 18 huoltokeskusta itselleen omilla yksiköillään, joita kaikilla pelaajilla on pelin alussa kolme lukuun ottamatta Venäjää, jolla niitä on neljä. Jokaisella pelikierroksella kukin pelaaja antaa samanaikaisesti määräykset yksiköilleen, ja yksilön liikuttua provinssiin kyseinen pelaaja hallitsee tätä provinssia.

Pelikierros koostuu neljästä vaiheesta: *Diplomatiavaiheesta*, jolloin pelaajat voivat neuvotella ja tehdä sopimuksia; *käskyjenkirjoitus-vaiheesta*, jolloin pelaajat antavat yksiköilleen käskyn joko pysyä paikallaan, liikkua tai tukea toista yksikköä; *käskyjen toteutusvaiheesta*, jolloin katsotaan, mitkä käskyt suoritetaan onnistuneesti; sekä *perääntymis- ja hajaantumisvaiheesta*, jossa pelaajien täytyy antaa perääntymiskäsky kaikille syrjäytetyille yksiköilleen. Joka toinen vuoro suoritetaan vuoron lopussa myös *yksikköjen päivitys*, jolloin pelaajat joilla on hallussaan enemmän huoltokeskuksia kuin yksiköitä saavat (tai menettävät) erotuksen verran yksiköitä. Käskyjen toteutusvaihe on säännöiltään vaiheista monimutkaisin, eikä sitä tässä kuvata tarkemmin. (Theodoridis ja Chalkiadakis, 2020)

Theodoridis ja Chalkiadakis (2020) toteuttivat peliin koettaan varten kahdeksan erilaista MCTS-tekoälytoimijaa. Koska kokeen tarkoitus on tutkia ainoastaan toimijoiden strategista pätevyyttä, rajattiin neuvotteluvaihe näiden toimijoiden osalta kokonaan pois. Jokainen toimija pyrkii maksimoimaan omistettujen huoltokeskusten määrän, mutta niiden toteutuksissa on vaihtelua käytetyn heuristiikan suhteen. Karkeasti nämä kahdeksan toimijaa voidaan jakaa kahteen ryhmään: Toimijat 1, 2, 3 ja 4 käyttävät hieman vaihtelevia heuristisia metodeja toiminnassaan, kun taas toimija 5 arvioi MCTS-algoritmin laajennusvaiheessa jokaista toimintoa solmussa käyttäen tarkoitukseen kehitettyä painotusjärjestelmää. Tämä järjestelmä laskee painotuksen jokaiselle

pelikartan alueelle, ja sitten laskee jokaisen toiminnon arvon perustuen tähän painotukseen. Laajennettavaksi valitaan korkeimman arvon saanut toiminto. Toimijat 6, 7 ja 8 pohjautuvat toimijaan 5, mutta muokkaavat sen toimintaa vaihtelevin tavoin.

Theodoridis ja Chalkiadakis (2020) pelauttivat MCTS-toimijoitaan kolmea erilaista tekoälyä vastaan. Ensimmäinen näistä oli nimeltään *satunnaisbotti* (RandomBot), joka päättää jokaisen toimintonsa täysin satunnaisesti. Muut kokeessa käytetyt tekoälyt eivät pelaa diplomatiavaihetta, mutta satunnaisbotit voivat neuvotella muiden satunnaisbottien kanssa satunnaisilla valinnoilla. Toinen kokeessa käytetty ja nimestään huolimatta kilpailukykyisempi tekoälytoimija oli *typeräbotti* (DumbBot). Se laskee jokaiselle alueelle kokonaisarvon, ja päättää käskyn jokaiselle yksikölleen näihin arvoihin perustuen. Myös aiemmin kuvattujen jälkimmäisen ryhmän MCTS-toimijoiden aluearviointi perustuu tälle ratkaisulle. Kolmas ja kahteen muuhun verrattuna vahvin kokeessa käytetty tekoäly oli suosittu ja aiempiin Diplomacy-tekoälyihin nähden ylivoimainen *D-Brane*, joka käyttää toiminnassaan laajaa joukkoa aluetietämykselle perustuvia heuristiikkoja.

Satunnaisbotteja vastaan MCTS-toimijoista sekä ensimmäinen ryhmä (toimijat 1, 2, 3 ja 4) että toinen ryhmä (toimijat 5, 6, 7 ja 8) pelasivat voitokkaasti. Ensimmäisen ryhmän tapauksessa etu ei ollut merkittävä, ja D-Brane pärjäsi satunnaisbotteja vastaan huomattavasti paremmin. Toinen MCTS-ryhmä suoriutui huomattavasti vahvemmin, jopa D-Braneen verrattuna. Typeräbotteja vastaan ensimmäisen ryhmän MCTS-toimijat pelasivat tappiollisesti, siinä missä toisen ryhmän MCTS-toimijat taas suoriutuivat voitokkaasti, joskaan eivät aivan yhtä hyvin kuin D-Brane. (Theodoridis ja Chalkiadakis, 2020)

D-Brane:a vastaan toisen ryhmän MCTS-toimijat menestyivät jotakuinkin sitä paremmin, mitä vähemmän pelissä oli D-Brane-pelaajia. Kaikki toisen ryhmän MCTS-toimijat kykenivät voittamaan D-Brane:n yksi vastaan yksi -turnauksissa, mutta lähes poikkeuksetta suoriutuivat tappiollisesti useamman pelaajan peleissä silloin, kun turnauksessa oli mukana kolme tai useampi D-Brane-pelaajaa. Tällöin ne eivät kyenneet hyväksikäyttämään heikompia toimijoita yhtä menestyksekkäästi. Siitä huolimatta ne pärjäsivät kilpailukykyisesti myös tällaisissa turnauksissa. Yleisesti huomataan, että Diplomacy-pelissä korkeatasoisen aluetietämyksen yhdistäminen MCTS-algoritmiin, kuten toisen ryhmän toimijoiden tapauksissa, vaikuttaisi lisäävän huomattavasti MCTS-pohjaisen toimijan suorituskykyä. (Theodoridis ja Chalkiadakis, 2020)

5.3 Settlers of Catan

Klaus Teuberin kehittämä Settlers of Catan on erittäin suosittu ja tunnettu usean pelaajan strateginen lautapeli. Settlers of Catanissa pelaajat asuttavat asuttamatonta saarta, josta he keräävät *resursseja* rakentaakseen uusia *asutuksia*, *kaupunkeja* ja *teitä*. Pelin voittaa pelaaja, joka kerää ensimmäisenä kymmenen *voittopistettä* (vp). Pelialue eli saari rakennetaan satunnaisesti 19 kuusikulmiosta siten, että se muodostaa yhden suuren kuusikulmion. Jokainen alue (kuusikulmio) yhtä aavikkoaluetta lukuun ottamatta tuottaa jotakin resurssia, ja pelaajilla on pelin alussa kaksi asutusta ja kaksi tietä valmiiksi rakennettuna. Pelaajat voivat rakentaa asutuksia ja kaupunkeja alueiden kulmapisteisiin ja teitä niiden reunoille. Asutusta ei kuitenkaan saa rakentaa sellaiseen kulmaan, joka on tasan yhden reunan matkan päässä toisesta asutuksesta tai kaupungista. (Szita et al. 2010)

Pelaajan vuoro alkaa *tuotantovaiheella*, jossa tämä heittää kahta noppaa. Noppien silmälukujen summa määrittää, mikä alue tuottaa vuorolla resursseja. Jokainen pelaaja, jolla on asutus tämän alueen reunalla, saa yhden alueen tuottamaa resurssia, kaupungin tapauksessa kaksi. Mikäli silmälukujen summa on seitsemän, ryöväri aktivoituu. Tällöin jokaisen yli seitsemän resurssikortin omistavan pelaajan täytyy heittää puolet niistä pois, ja vuorossa oleva pelaaja siirtää ryövärin uudelle alueelle. Tämä alue tulee *blokatuksi*, eikä tuota mitään. Tuotantovaiheen jälkeen pelaaja voi rakentaa resurssikorteillaan uusia asutuksia ja teitä, tai päivittää asutuksia kaupungeiksi. Lisäksi pelaajat voivat ostaa resurssikorteillaan satunnaisen *kehityskortin*, joka voi antaa voittopisteen tai vaihtelevia bonuksia. Voittopisteitä saa kehityskorttien lisäksi myös asutuksista (1 vp), kaupungeista (2 vp) sekä joistakin *voitonmerkeistä* joita pelissä voi saada haltuunsa, esimerkiksi olemalla ensimmäinen pelaaja joka rakentaa viiden tien ketjun. (Szita et al. 2010)

Szita et al. (2010) kehittivät MCTS-pohjaisen tekoälyn, *SmartSettlerin*, Settlers of Cataniin. Toteutussyistä he poistivat toteutuksestaan epätäydellisen tiedon elementit, eli vastustajien kehityskorttien simuloinnin, sillä heidän mukaansa niillä ei ole suurta merkitystä pelaajan strategian kannalta. Lisäksi heidän toimijansa ei käy kauppaa, mutta kokeeseen osallistuvat muut tekoälytoimijat voivat toki vapaasti tehdä niin keskenään. He tutkivat myös pelaajien vuorojärjestyksen vaikutusta ja havaitsivat merkittäviä tilastollisia eroja, joten varsinaisissa kokeissaan he satunnaistivat tekoälyjen vuorojärjestyksen. SmartSettleriin injektoitiin myös hieman aluetietämystä: kaikki asutuksenrakennustoiminnot saavat simulaatioissa 20 ”virtuaalivoittoa” (virtual wins), ja kaikki kaupunginrakennukset 10. Tällöin kyseisten toimintojen tullessa lisätyksi hakupuuhun, sen käyntikertojen ja voittokertojen määräksi asetetaan 20 tai 10, vastaavasti. Näitä

virtuaalivoittoja ei kuitenkaan propagoida takaisin ylemmille solmuille, sillä se vääristäisi valintaa huomattavasti. (Szita et al. 2010)

Szita et al. (2010) pelauttivat SmartSettlersiä sekä heuristista *JSettlers*-tekoälyä että ihmispelaajia vastaan. SmartSettlersin MCTS-simulaatioiden määrä jokaista siirtoa kohden oli vaihtelevasti joko 1 000 tai 10 000. JSettlers-tekoälyä vastaan pelatessa pelissä oli mukana yksi SmartSettlers ja kolme JSettlers-toimijaa. 1 000:n simulaation SmartSettlers suoriutui vain marginaalisesti paremmin kuin JSettlers - sen voittoprosentti neljän pelaajan peleissä oli 27%. Suuremman simulaatiomäärän SmartSettlers puolestaan oli ratkaisevasti JSettlersiä vahvempi, voittaen 49% kaikista peleistä ja keräten runsaasti pisteitä myös silloin, kun se ei voittanut.

Ihmispelaajia vastaan pelattujen pelien määrä ei ollut riittävän suuri, jotta siitä voitaisiin johtaa tilastollisesti päteviä päätelmiä. Laadullisesti Szita et al. (2010) kuitenkin arvioivat, että SmartSettlers kykeni tekemään järkeviä siirtoja, jotka monessa tilanteessa täsmäsivät kokeneen ihmispelaajan siirtoihin. Kokenut ihmispelaaja kykeni kuitenkin yleensä voittamaan SmartSettlerin. Szita el al. (2010) uskovat, että simulaatiomäärän lisääminen tai siirronvalintaheuristiikan parantaminen voisi poistaa tai lieventää joitakin SmartSettlersin heikkouksia, vaikkakin ainakin edellisessä tapauksessa nopeuden kustannuksella. Toimijan onnistuminen Settlers of Catanissa osoitti kuitenkin Szitan et al. (2010) mielestä sen, että MCTS voi olla pätevä lähestymistapa myös monimutkaisempiin moninpeleihin.

5.4 AI Factory Spades

Spades on AI Factory:n kehittämä suosittu mobiilikorttipeli, jossa ihmispelaaja pelaa kahta tekoälyvastustajaa vastaan yhden tekoälykumppanin kanssa. Peli pohjautuu nimensä mukaisesti *Spades-tikkikorttipelille* (trick taking card game), joka muistuttaa hieman bridge-korttipeliä. Peliä pelataan useita kierroksia, ja jokaisen kierroksen lopussa kukin joukkue pisteytetään. Peli päättyy kun yksi tai useampi joukkue on saanut yhteensä 500 pistettä, jolloin korkeamman pistemäärän saanut joukkue voittaa. (Pagat, 2013)

Jokaisen kierroksen alussa kullekin pelaajalle jaetaan 13 korttia tavallisesta korttipakasta. Jokainen pelaaja asettaa *panoksen*, joka on arvio siitä, kuinka monta *tikkiä* he uskovat voittavansa kyseisellä kierroksella. Tämän jälkeen jokainen pelaaja pelaa vuorollaan yhden kortin pöytään. Yksi pelaajista on *johtaja*, joka voi pelata minkä tahansa kortin. Seuraavien pelaajien täytyy vastata tähän samaa

maata olevalla kortilla, mikäli heillä sellainen on; muussa tapauksessa he saavat pelata minkä tahansa kortin. Tikin voittava kortti on korkein johtajan asettamaa maata oleva kortti, ellei joku pelaa patakorttia, jotka toimivat pelissä valttikortteina. Tällöin korkein patakortti voittaa tikin. Kunkin tikin voittajasta tulee seuraavan tikin johtaja. (Pagat, 2013)

Koska käsikortteja on 13, kukin kierros koostuu 13 tikistä. Kierroksen lopussa lasketaan, kykenikö kukin joukkue saamaan yhtä paljon tai enemmän tikkejä kuin heidän yhteenlaskettu panoksensa. Mikäli he onnistuivat, saavat he kymmenen kertaa panoksensa määrän pisteinä - muussa tapauksessa he menettävät vastaavan määrän. Panoksesta ylimenevät tikit antavat joukkueelle *reppuja* (bags), jotka ovat yhden pisteen arvoisia. Jos reppuja kerääntyy pelin aikana yli 10, joukkue menettää 10 reppua ja 100 pistettä. Pelaajat voivat panostaa myös nolla tikkiä, jolloin säännöt ovat hieman erilaiset: Jos kyseinen pelaaja ei voittanut yhtään tikkiä, joukkue saa 100 pistettä, ja muussa tapauksessa se menettää 100 pistettä. (Pagat, 2013)

Whitehouse et al. (2013) kehittivät AI Factory:n Spades-peliin MCTS-pohjaisen tekoälyn osittain korvaamaan pelin aiempaa tietämyspohjaista tekoälyä. Toteutuksen tarkoituksena oli parantaa sekä tekoälyn vahvuutta korkeamman tason ihmispelaajien kanssa pelatessa, että myös tekoälyn *havaittua* vahvuutta eli sitä, kuinka pätevänä ihmispelaajat tekoälyn päätöksiä pitävät. Heidän toteutuksensa käyttää MCTS:n *ISMCTS* (*Information Set Monte Carlo Tree Search*) -muunnelmaa. ISMCTS ottaa huomioon kätketyn tiedon muodostamalla pelitilasta jokaiseen simulaatioon satunnaisen *determinisaation*. Determinisaatio-tekniikkaa hyödynnettiin myös aiemmin kuvatuissa Cowlingin et al. (2012) MTG-ratkaisuissa. Kukin determinisaatio vastaa yhtä mahdollista pelitilaa myös kätketyn tiedon, Spade-pelin tapauksessa muiden pelaajien käsikorttien, osalta. Tällöin myös epätäydellisen tiedon peleissä voidaan rakentaa vain yksi MCTS-puu, joka kerää tilastollista tietoa monista eri determinisaatioista. Whitehousen et al. (2013) toteutus hyödyntää puuhaussa myös heuristista tietämystä, mutta siten, että haun edetessä tietämyksen painotus vähenee lopulta nollaan, jolloin heuristiikka ei liiaksi jyrää MCTS:n toimintaa.

Pelikokeissa Whitehousen et al. (2013) MCTS-toimija osoittautui huomattavasti vahvemmaksi kuin oletustasoinen tietämyspohjainen tekoäly. Yli 2 600 simulaatiota siirtoa kohden suorittavat MCTS-versiot osoittautuivat vähintään yhtä vahvoiksi kuin tietämyspohjaisen tekoälyn haastavin versio, joka tarvitsi yli kaksi sekuntia jokaiseen siirtoon Samsung Galaxy S II -puhelimella. Vertailun vuoksi samalla laitteella 5 000 simulaation MCTS-toimija käytti siirtoa kohden vain 0,5 sekuntia. Heuristisen tietämyksen lisääminen MCTS-toimijoille ei näyttänyt aiheuttavan tilastollisesti

merkittävää parannusta. Taktisesti erityislaatuisissa nollapanostuksen peleissä MCTS-toteutus osoittautui huomattavasti vahvintakin käytettyä tietämyspohjaista tekoälyä kyvykkäämmäksi, siitäkin huolimatta että tietämyspohjainen ratkaisu käytti monia erillisiä nollapanostusheuristiikkoja MCTS-toteutuksen toimiessa täsmälleen samalla tavalla kuin normaalistikin.

Peliteoreettisesti optimaalinenkaan tekoälytoimijan peli ei kuitenkaan takaa ihmispelaajien tyytyväisyyttä silloin, kun pelaajan tekoälyjoukkuetoveri tekee siirtoja, jotka vaikuttavat ihmispelaajan intuition pohjalta huonoilta. Baier et al. (2019) jatkokehittivät edellä kuvattua Spades-pelin ISMCTS-tekoälyä ihmismäisempään suuntaan, korvaten aiemman aluetietämyksen neuroverkolla, joka ohjaa MCTS:n toimintaa pelaajamallinnuksen pohjalta. He osoittivat empiirisesti, että *epäsuora imitaatio* (indirect imitation), joka yhdistää neuroverkon ennusteen puuhaun kanssa, on ylivertainen verrattuna *suoraan imitatioon* (direct imitation), jossa neuroverkon ennuste pelataan suoraan. Heidän parannuksensa kykeni aiempaa toteutusta ihmismäisempään peliin säilyttäen silti ISMCTS:n kilpailukykyisen pelitason, ilman laskennallista lisäkuormaa. (Baier et al. 2019)

5.5 Hearthstone

Suosittu elektroninen keräilykorttipeli Hearthstone on hyvin monilta osiltaan erittäin paljon luvussa 4 kuvatun Magic: the Gathering -keräilykorttipelin kaltainen. Pelejä on usein verrattu toisiinsa, ja MTG voidaankin nähdä Hearthstonen edeltäjänä, vaikka molempia pelataan aktiivisesti edelleen. Tekoälyn kannalta olennaisilta piirteiltään pelit ovat yhdenmukaisia: Kummatkin ovat epätäydellisen informaation stokastisia pelejä, joissa on laaja määrä erilaisia kortteja omine vaikutuksineen. Suurimpana erona voidaan pitää sitä, että toisin kuin MTG:ssä, Hearthstonessa pelaajat eivät voi vuorovaikuttaa vastustajansa kanssa tämän vuorolla. Lisäksi MTG:ssä olioiden määrää pelialueella ei ole rajoitettu, eikä manakapasiteetin kasvaminen vuorojen edetessä ole taattu. Hearthstonessa voidaan myös katsoa olevan MTG:tä enemmän stokastisia elementtejä.

Wang ja Moh (2019) kehittivät Hearthstoneen MCTS-pohjaisen tekoälytoimijan nimeltään *Pyra*. Jo aiemmin järjestetyssä Hearthstone-aiheisessa AAIA'17 tiedonlouhintahaasteessa MTCS oli todettu optimaaliseksi tavaksi simuloida Heartstone-pelejä (Janusz et al. 2017). Hearthstonen stokastiset elementit tekevät kuitenkin Wangin ja Mohin (2019) mukaan puuhakupohjaisista lähestymistavoista mahdottoman raskaita toteuttaa vakiomuotoisella MCTS-algoritmilla. Tämän ongelman ratkaisemiseksi he rajoittivat puuhaun 15 leveyteen ja 5 syvyyteen, sekä eliminoivat siitä

tarpeettomat välivaiheet, keskittyen vain vuorojen lopputiloihin. Nämä keinot eivät kuitenkaan yksinään riittäneet, vaan myös pelitilan evaluaatiolle oli tarvetta. Tähän tarkoitukseen käytettiin syvää neuroverkkoa, joka koulutettiin käyttämällä silloista HSReplay-sivuston (HSReplay, 2023) pelattujen pelien tietokantaa.

Wang ja Moh (2019) pelauttivat Pyraa kuutta eri tekoälyä vastaan. Viisi niistä oli manuaalisesti luotuja heuristisia tekoälyratkaisuja, ja kuudes puolestaan Pyra:n kanssa samankaltainen MCTS-ratkaisu sillä erolla, että neuroverkon sijasta se käytti lineaariseen regressioon pohjautuvaa oppijaa pelitilan evaluaatiossa. Pyra pelasi 1 000 peliä jokaista tekoälyä vastaan, ja saavutti kaikkia manuaalisia heuristisia tekoälyjä vastaan pienimmillään 77% ja suurimmillaan 88% voittoprosentit. MCTS:ää ja lineaarista regressiota käyttävää tekoälyä vastaan etu oli pienempi mutta silti merkittävä, 69% voittoprosentilla. (Wang ja Moh, 2019)

6 Pohdintaa ja johtopäätökset

Tässä tutkielmassa käytiin läpi kirjallisuuskatsauksen keinoin joukko vuoropohjaisia strategiapelejä. Tarkoituksena oli tutustua peleihin sovellettuihin tekoälyratkaisuihin, ja etenkin sellaisiin, jotka hyödynsivät Monte Carlo -puuhakua (MCTS). Shakin kohdalla nähtiin, että jo MCTS-puuhakua edeltävät minimax- ja aluetietämyspohjaiset ratkaisut, kuten Deep Blue ja Stockfish, kykenivät haastamaan huipputason ihmispelaajia ja Stockfishin tapauksessa jopa jättämään heidät lopulta kauas taakseen. MCTS- ja neuroverkkopohjainen AlphaZero osoittautui kuitenkin jopa Stockfishia vahvemmaksi, vaikka ei hyödynnä toteutuksessaan lainkaan pelikohtaista aluetietämystä.

Go-peliä puolestaan on historiallisesti pidetty huomattavasti shakkia haastavampana kohteena päteville tekoälyratkaisuille. Tutkielmassa nähtiin, että vasta MCTS-pohjaisten ratkaisuiden myötä go-tekoälyt alkoivat kehittyä tarpeeksi vahvoiksi haastaakseen kärkiluokan ihmispelaajat. Kehitys saavutti eräänlaisen huippunsa, kun MCTS-pohjainen AlphaGo voitti kaksi maailman kärkeä edustavaa go-ihmispelaajaa - saavutus, jota monet pitkään pitivät pelin kohdalla mahdottomana.

Pokeri ja Magic: the Gathering -keräilykorttipeli (MTG) eroavat tekoälyratkaisuiden näkökulmasta shakista ja go-pelistä ei-deterministisen luonteensa ja epätäydellisen tietonsa vuoksi. Nämä tekijät lisäävät myös huomattavasti pelien haasteellisuutta tekoälyratkaisuiden suhteen, sillä satunnaisuustekijät ja epätäydellinen tieto laajentavat perinteisten puuhakuratkaisuiden hakuavaruutta kohtuuttomasti. Tutkielmassa havaittiin, että MCTS saattaa osoittautua vahvaksi kehitysväyläksi myös tällaisten aiemmille tekniikoille ongelmallisten pelien kohdalla. Lähellekään pätevää täysipainoista MTG-tekoälyä ei tähän päivään mennessä ole kehitetty, mutta tässä tutkielmassa tarkastellut ratkaisut ovat askeleita lupaavaan suuntaan.

Lopuksi tarkasteltiin myös joukkoa MCTS-ratkaisuja muissa vuoropohjaisissa strategiapeleissä. Kaikki kyseisessä luvussa tarkastellut pelit sisälsivät joko stokastisuutta, kätkettyä tietoa, tai molempia. Lisäksi monet niistä olivat useamman kuin kahden pelaajan pelejä, joista löytyy runsaasti strategista syvyyttä. Perinteisten puuhakuratkaisuiden rajojen voidaan katsoa tulevan näiden pelien kohdalla erittäin nopeasti vastaan. Havaittiin, että MCTS-pohjaiset ratkaisut kykenivät näiden pelien kohdalla lähes poikkeuksetta parempaan suoritukseen kuin perinteisemmät, aluetietämykseen perustuvat toimijat. Voidaankin todeta, että MCTS vaikuttaisi tarjoavan erinomaisen ratkaisupohjan sellaisten pelien tekoälyratkaisuille, joiden kohdalla perinteisemmät

tekniikat ovat osoittautuneet laskentatehollisista tai muista syistä riittämättömiksi. Etenkin pelit, joiden pelitilan suotuisuutta on hankalaa arvioida aluekohtaiseen tietämykseen nojaavalla evaluaatiofunktiolla, hyötyvät erityisen paljon MCTS:n yleisluontoisesta lähestymistavasta.

Jatkotutkimuksen kannalta saattaisi olla mielenkiintoista perehtyä Monte Carlo -puuhaun sovelluksiin myös reaaliaikaisten pelien tekoälyratkaisuiden osalta. Lisäksi tarkempi ja kapeampi perehtyminen erityisesti runsasta stokastisuutta ja kätkettyä tietoa sisältävien pelien MCTS-toteutuksiin voisi mahdollisesti olla hedelmällinen aihe jatkotutkimukselle.

Lähdeluettelo

Campbell, M. S., & Marsland, T. A. (1983). A Comparison of Minimax Tree Search Agorithms. *Artificial Intelligence*, *20*(4), 347-367.

Russell, S. and Norvig, P.. *Artificial Intelligence : a Modern Approach*. 3rd edition. Boston: Pearson, 2016.

Fuller, S. H., Gaschnig, J. G., & Gillogly, J. J. (1973). *Analysis of the Alpha-Beta Pruning Algorithm*. Department of Computer Science, Carnegie-Mellon University.

Chaslot, G. M. J., Winands, M. H., Herik, H. J. V. D., Uiterwijk, J. W., & Bouzy, B. (2008). Progressive strategies for Monte-Carlo tree search. *New Mathematics and Natural Computation*, *4*(03), 343-357.

Kocsis, L., & Szepesvári, C. (2006). Bandit based Monte-Carlo Planning. In *Machine Learning: ECML 2006: 17th European Conference on Machine Learning Berlin, Germany, September 18-22, 2006 Proceedings 17* (pp. 282-293). Springer Berlin Heidelberg.

Gelly, S., & Silver, D. (2011). Monte-Carlo tree search and rapid action value estimation in computer Go. *Artificial Intelligence*, *175*(11), 1856-1875.

Kitchenham, B., & Charters, S. (2007). Guidelines for performing Systematic Literature Reviews in Software Engineering.

Campbell, M., Hoane Jr, A. J., & Hsu, F. H. (2002). Deep Blue. *Artificial intelligence*, *134*(1-2), 57-83.

Maharaj, S., Polson, N., & Turk, A. (2022). Chess AI: Competing Paradigms for Machine Intelligence. *Entropy*, *24*(4), 550.

Companez, N., & Aleti, A. (2016). Can Monte-Carlo Tree Search learn to sacrifice?. *Journal of Heuristics*, *22*(6), 783-813.

Xie, F., & Liu, Z. (2009, November). Backpropagation Modification in Monte-Carlo Game Tree Search. In *2009 Third International Symposium on Intelligent Information Technology Application* (Vol. 2, pp. 125-128). IEEE.

Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., ... & Hassabis, D. (2018). A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, *362*(6419), 1140-1144.

Gelly, S., Kocsis, L., Schoenauer, M., Sebag, M., Silver, D., Szepesvári, C., & Teytaud, O. (2012). The Grand Challenge of Computer Go: Monte Carlo Tree Search and Extensions. *Communications of the ACM*, *55*(3), 106-113.

Cai, X., & Wunsch, D. C. (2007). Computer Go: A grand challenge to AI. *Challenges for Computational Intelligence*, 443-465.

Ryder, J. L. (1971). *Heuristic Analysis of Large Trees as Generated in the Game of Go* (No. 155). Stanford University.

Brügmann, B. (1993). *Monte Carlo Go* (Vol. 44). Syracuse, NY: Technical report, Physics Department, Syracuse University.

Cazenave, T., & Helmstetter, B. (2005). Combining Tactical Search and Monte-Carlo in the Game of Go. *CIG*, *5*, 171-175.

Gelly, S., Wang, Y., Munos, R., & Teytaud, O. (2006). *Modification of UCT with patterns in Monte-Carlo Go* (Doctoral dissertation, INRIA).

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., ... & Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *nature*, *529*(7587), 484-489.

Billings, D., Davidson, A., Schaeffer, J., & Szafron, D. (2002). The Challenge of Poker. *Artificial Intelligence*, *134*(1-2), 201-240.

Rubin, J., & Watson, I. (2011). Computer poker: A review. *Artificial intelligence*, *175*(5-6), 958-987.

Schweizer, I., Panitzek, K., Park, S. H., & Fürnkranz, J. (2009). An Exploitative Monte-Carlo Poker Agent. In *KI 2009: Advances in Artificial Intelligence: 32nd Annual German Conference on AI, Paderborn, Germany, September 15-18, 2009. Proceedings 32* (pp. 65-72). Springer Berlin Heidelberg.

Van den Broeck, G., Driessens, K., & Ramon, J. (2009). Monte-Carlo Tree Search in Poker Using Expected Reward Distributions. In *Advances in Machine Learning: First Asian Conference on Machine Learning, ACML 2009, Nanjing, China, November 2-4, 2009. Proceedings 1* (pp. 367-381). Springer Berlin Heidelberg.

Heinrich, J., & Silver, D. (2015, July). Smooth UCT Search in Computer Poker. In *IJCAI* (pp. 554-560).

Wizards of the Coast. 2023. Magic: the Gathering Comprehensive Rules. https://magic.wizards.com/en/rules

Ward, C. D., & Cowling, P. I. (2009, September). Monte Carlo Search Applied to Card Selection in Magic: The Gathering. In *2009 IEEE Symposium on Computational Intelligence and Games* (pp. 9-16). IEEE.

Bjørke, S. J., & Fludal, K. A. (2017). *Deckbuilding in Magic: The Gathering Using a Genetic Algorithm* (Master's thesis, NTNU).

Ward, H. N., Mills, B., Brooks, D. J., Troha, D., & Khakhalin, A. S. (2021, August). AI solutions for drafting in Magic: the Gathering. In *2021 IEEE Conference on Games (CoG)* (pp. 1-8). IEEE.

Cowling, P. I., Ward, C. D., & Powley, E. J. (2012). Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. *IEEE Transactions on Computational Intelligence and AI in Games*, *4*(4), 241-257.

Ameneyro, F. V., Galván, E., & Morales, Á. F. K. (2020, December). Playing Carcassonne with Monte Carlo Tree Search. In *2020 IEEE Symposium Series on Computational Intelligence (SSCI)* (pp. 2343-2350). IEEE.

Theodoridis, A., & Chalkiadakis, G. (2020, September). Monte Carlo Tree Search for the Game of Diplomacy. In *11th Hellenic Conference on Artificial Intelligence* (pp. 16-25).

Szita, I., Chaslot, G., & Spronck, P. (2010). Monte-Carlo Tree Search in Settlers of Catan. In *Advances in Computer Games: 12th International Conference, ACG 2009, Pamplona Spain, May 11-13, 2009. Revised Papers 12* (pp. 21-32). Springer Berlin Heidelberg.

Pagat. (2013). Spades-pelin säännöt. https://www.pagat.com/auctionwhist/spades.html.

Whitehouse, D., Cowling, P., Powley, E., & Rollason, J. (2013). Integrating Monte Carlo Tree Search with Knowledge-Based Methods to Create Engaging Play in a Commercial Mobile Game. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* (Vol. 9, No. 1, pp. 100-105).

Baier, H., Sattaur, A., Powley, E. J., Devlin, S., Rollason, J., & Cowling, P. I. (2018). Emulating Human Play in a Leading Mobile Card Game. *IEEE Transactions on Games*, *11*(4), 386-395.

HSReplay. (2023). Pelitietokanta. https://hsreplay.net

Wang, D., & Moh, T. S. (2019, April). Hearthstone AI: Oops to Well Played. In *Proceedings of the 2019 ACM Southeast Conference* (pp. 149-154).