Emil Keränen

Vertaileva tutkimus koneoppimisen hyödyntämisestä videopelien reitinhaussa

Tietotekniikan pro gradu -tutkielma

30. syyskuuta 2022

Jyväskylän yliopisto

Informaatioteknologian tiedekunta

Tekijä: Emil Keränen

Yhteystiedot: emil.a.keranen@student.jyu.fi

Ohjaaja: Tommi Kärkkäinen

Työn nimi: Vertaileva tutkimus koneoppimisen hyödyntämisestä videopelien reitinhaussa

Title in English: Comparative study of utilizing machine learning in video games' pathfinding

Työ: Pro gradu -tutkielma

Opintosuunta: Tietotekniikka

Sivumäärä: 33+0

Tiivistelmä: TODO: tiivistelmä suomeksi

Avainsanat: koneoppiminen, videopeli, reitinhaku, syvä vahvistusoppiminen

Abstract: TODO: In english

Keywords: machine learning, video game, pathfinding, deep reinforcement learning, Soft Actor Critic, Machine Learning Agents, Unity

Kuviot

Sisällys

1 Johdanto

- Tutkimuksen kohteena koneoppimisen tehostama reitinhaku videopeleissä ja sen vertaaminen heuristiseen A*-algoritmiin.

- Yleisesti reitinhaulla tarkoitetaan alku- ja loppupisteen välisen reitin selvittämistä. Useimmiten tarkoituksena on löytää lyhin reitti väistellen samalla matkan varrella olevia esteitä.

- Reitinhakua tarvitaan videopelien lisäksi myös mm. robotiikassa.

- Videopeleissä reitinhaku ilmenee pääasiassa tekoälyagenttien suorittamana toimintana, joten tässä tutkimuksessa keskitytään agentteihin. Näitä agentteja kutsutaan myös ei-pelaaja-hahmoiksi (engl. non-player-character, NPC).

- Ei-pelaaja-hahmot ja itseasiassa videopelien reitinhaku vertautuvat hyvin robotiikkaan ja robottien reitinhakuun.

- Sekä robotiikassa että videopeleissä toiminta-alue voi muuttua hyvinkin paljon reaaliajassa, jolloin reitinhaun täytyy sopeutua muutoksiin nopeasti. Käytetyt ratkaisut sen sijaan voivat vaihdella näiden kahden osa-alueen välillä: robotiikassa tarkkuus ja turvallisuus nousevat tärkeimmiksi ominaisuuksiksi ja vastaavasti videopeleissä nopeus määrittää reitinhaun "hyvyyden".

- Reitinhaku on aina ollut vaativa ongelma videopeleissä, mutta nykyään reitinhaun ongelmallisuus voidaan useimmissa tapauksissa sivuuttaa laatimalla heuristinen ratkaisu A*-algoritmin avulla.

- Koneoppiminen mahdollistaa aiemman kokemuksen hyödyntämisen myöhemmässä toiminnassa. Agentteja voidaan kouluttaa harjoitteludatan avulla, jolloin ne oppivat toimimaan tuntemattomissa tilanteissa. Koneoppimisen ansiosta reitinhaku-agentti voidaan opettaa toimimaan vaativissa ja dynaamisissa pelialueissa, joissa muuttuvat esteet ja alueen labyrinttimäisyys heikentävät A*-algoritmin toimintaa.

- Tutkimuksen ideana on käyttää Unity-pelimoottorille luotuja koneoppimisagentteja (engl. Unity Machine Learning Agents) ja opettaa niitä erilaisten pelialueiden avulla. Opettamisen

jälkeen agentteja testataan oikeilla pelialueilla ja verrataan tuloksia A*-algoritmilla saatuihin tuloksiin.

- ML-agents perustuu PyTorch-kirjastoon ja mahdollistaa vahvistusoppimisen hyödyntämisen.

- Tensorboard-lisäosan avulla voidaan visualisoida palkkioiden keskiarvot ja opetuksen edistyminen opetuksen aikana.

- Pelialueiden on tarkoitus olla monimutkaisia ja dynaamisia, koska A*-algoritmi suoriutuu yksinkertaisista reitinhakutehtävistä moitteettomasti.

2 Reitinhaku videopeleissä

Reitinhaulla tarkoitetaan yksinkertaisimmillaan reitin tai polun selvittämistä kahden pisteen välillä. Se on yksi videopelien tekoälyn ja myös robotiikan tunnetuimmista ja haastavimmista ongelmista, jota on tutkittu jo muutaman vuosikymmenen ajan (Cui ja Shi 2011; Abd Algfoor, Sunar ja Kolivand 2015). Reitinhakua esiintyy monissa eri peligenreissä, kuten roolipeleissä ja reaaliaikaisissa strategiapeleissä, joissa ei-pelaaja-hahmoja (engl. non-player-character, NPC) määrätään liikkumaan ennaltamäärättyyn tai pelaajan määräämään sijaintiin väistellen samalla vastaantulevia esteitä. Reitinhaun ja yleisesti tekoälyn on oltava realistista, jotta pelaaja pystyy syventymään videopelin maailmaan eikä pelaajan kokema immersio keskeydy.

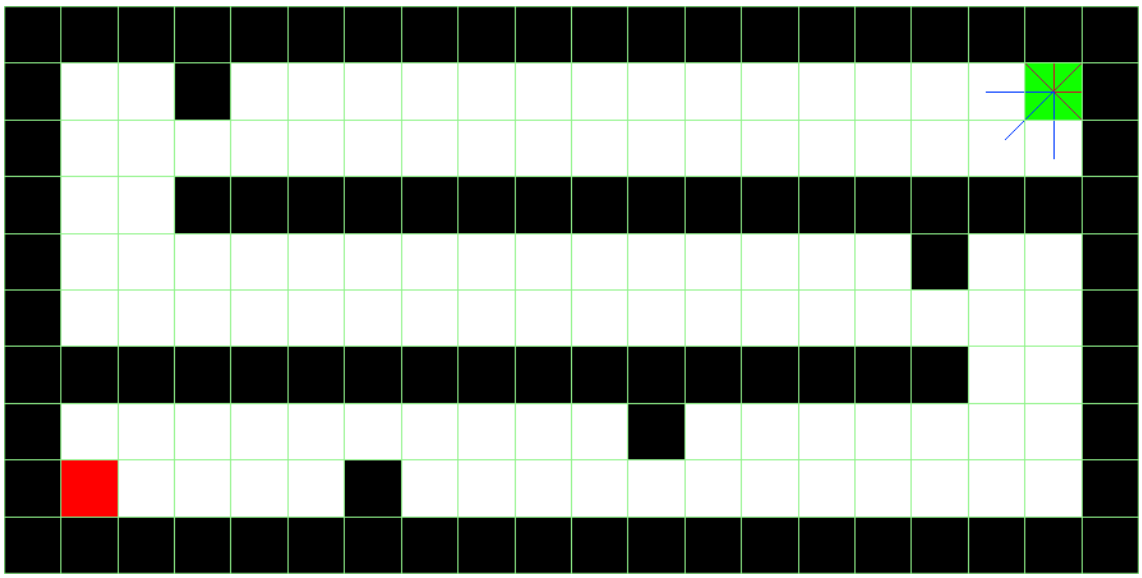
2.1 Pelialueen esitystavat

Pelialue esitetään reitinhakua varten aina graafina, joka koostuu solmuista ja kaarista. Kaaret yhdistävät solmut toisiinsa ja mahdollistavat liikkumisen solmujen välillä. Graafista voidaan muodostaa erilaisia kokonaisuuksia, joita kutsutaan ruudukoiksi (engl. grid). Ruudukot voivat olla kuvioltaan säännöllisiä tai epäsäännöllisiä. (Lawande ym. 2022). Säännölliset ruudukot sisältävät esimerkiksi kolmioita, kuusikulmioita, neliöitä tai kuutioita riippuen onko kyseessä 2D- vai 3D-alue. Epäsäännölliset ruudukot voivat koostua esimerkiksi reittipisteistä (engl. waypoint) tai navigointiverkosta (engl. navigation mesh). Selvästi käytetyin esitystapa videopeleissä on 2D-neliöruudukko. (Lawande ym. 2022).

Yleisesti ruudukot sisältävät yksittäisiä vapaita ruutuja tai esteruutuja (engl. tile). Vapaat ruudut muodostavat graafin, jolloin jokainen vapaa ruutu vastaa yhtä graafin solmuista. Vierekkäisiä ruutuja yhdistävät kaaret, joita pitkin reitinhaku ja liikkuminen tapahtuu. (Botea ym. 2013). Solmujen vierekkäisyys voi tarkoittaa horisontaalista ja vertikaalista vierekkäisyyttä (neljä suuntaa) tai näiden lisäksi myös diagonaalista vierekkäisyyttä (kahdeksan suuntaa). (Abd Algfoor, Sunar ja Kolivand 2015; Botea ym. 2013). Kuva 1 havainnollistaa yksinkertaista ruudukkoaluetta, jossa liikkuminen tapahtuu neljään tai kahdeksaan suuntaan.

Vaikka neliöruudukkoa pidetään tunnetuimpana ja käytetyimpänä esitystapana, se voi kui-

tenkin osoittautua ongelmalliseksi tilanteessa, jossa hahmo voi liikkua diagonaalisesti, jolloin kaikki vierekkäiset ruudut eivät ole saman etäisyyden päässä toisistaan. Ruudukossa, jossa ruudut ovat kooltaan 1x1, diagonaalinen etäisyys on $\sqrt{2}$, kun taas vertikaalinen tai horisontaalinen etäisyys on 1. (TODO: tähän jotain lähdettä mahdollisista ongelmista)



Kuvio 1. Ruudukko, jossa mustat ruudut ovat esteruutuja ja valkoiset ruudut vapaita ruutuja. Vihreä ruutu on aloitusruutu ja punainen ruutu on maaliruutu.

Vähemmän tunnettuja pelialueen esitystapoja ovat kolmiointi ja kuusikulmiointi (Abd Algfoor, Sunar ja Kolivand 2015). Alueen kolmiointi ja siihen perustuvat TA*- ja TRA*-algoritmi ovat kuitenkin osoittautuneet moninkertaisesti nopeammiksi suurissa pelialueissa verrattuna A*-algoritmiin (Demyen ja Buro 2006). Kuusikulmioihin perustuvat alueet ja reitinhakualgoritmit ovat myös tuottaneet lupaavia tuloksia robotiikan tutkimuksessa sekä yleisesti suoriutuneet paremmin muistinkäytön ja ajankäytön suhteen verrattuna neliöruudukkoihin (Abd Algfoor, Sunar ja Kolivand 2015; Lawande ym. 2022). Epäsäännöllisistä ruudukoista navigointiverkkoa on käytetty suurimmaksi osin videopeleissä ja esimerkiksi Unity tarjoaa dokumentaatiossaan laajat ohjeet ja menetelmät navigaatioverkon luomiseen (Lawande ym. 2022; “Unity Docs” 2022).

2.2 Reitinhakualgoritmit

Graafin lyhyimmän polun ongelmaa ja eri reitinhakualgoritmeja on tutkittu jo vuosikymmenten ajan. Vanhimmat ja tunnetuimmat algoritmit, Dijkstran algoritmi (Dijkstra ym. 1959) ja A*-algoritmi (Hart, Nilsson ja Raphael 1968), esiteltiin jo 50- ja 60-luvuilla ja ne pystyivät ratkaisemaan lyhimmän polun ongelman staattisessa graafissa. Uudemmat reitinhaun sovellukset, kuten itseohjautuvat autot ja robotit, toivat kuitenkin alkuperäiselle ongelmanratkaisulle lisää vaatimuksia. Lyhimmän polun löytämisen lisäksi reitinhakualgoritmin täytyy ottaa huomioon sovelluksesta riippuen reitin turvallisuus, tehokkuus ja mahdollisten esteiden väistäminen (Karur ym. 2021).

A*-algoritmi on selvästi tunnetuin videopelien ja robottien reitinhaussa nopeutensa ansiosta (Cui ja Shi 2011; Abd Algfoor, Sunar ja Kolivand 2015; Botea ym. 2013). A*-algoritmista on kehitetty jo monia eri variaatioita, jotka pyrkivät vastaamaan jatkuvasti kasvaviin vaatimuksiin. Tässä tutkimuksessa keskitytään tarkemmin A*-algoritmiin ja sen variaatioihin. (TODO: tähän vielä muutoksia, kuulostaa toistolta)

2.2.1 A*-algoritmi

A*-algoritmi on hyvin tunnettu paras-ensin -reitinhakualgoritmi, joka hyödyntää heuristista arviointifunktiota lyhimmän reitin etsimiseen (Cui ja Shi 2011; Duchoň ym. 2014). A*-algoritmia voidaan pitää käytetyimpänä graafien etsintäalgoritmina etenkin videopeleissä (Botea ym. 2013; Lawande ym. 2022).

Algoritmin toiminta tapahtuu seuraavasti: jokainen aloitussolmun vierekkäinen solmu arvioidaan kaavan

$$f(n) = h(n) + g(n)$$

mukaisesti, jossa n on solmu, $h(n)$ on heuristinen etäisyys solmusta n maalisolmuun ja $g(n)$ on todellinen etäisyys aloitussolmusta solmuun n. Näistä solmuista matalimman $f(n)$-arvon solmu käsitellään seuraavaksi, jolloin kyseisen solmun vierekkäisten solmujen $f(n)$-arvot lasketaan. Tämä prosessi jatkuu, kunnes maalisolmu saavutetaan. Heuristiikan ollessa nolla A*-algoritmista tulee Dijkstran algoritmi.

A*-algoritmilla on kolme esitettyä ominaisuutta (Hart, Nilsson ja Raphael 1968). Ensiksi A*-algoritmi löytää reitin, jos sellainen on olemassa. Toiseksi reitti on optimaalinen, jos heuristiikka on luvallinen eli arvioitu etäisyys on lyhyempi tai yhtä suuri kuin todellinen etäisyys. Viimeisenä mikään muu algoritmi samalla heuristiikalla ei käy läpi vähemmän solmuja kuin A*-algoritmi eli A* käyttää heuristiikkaa tehokkaimmalla mahdollisella tavalla. (Hart, Nilsson ja Raphael 1968; Cui ja Shi 2011). Luvallisia heuristiikkoja ovat solmujen vierekkäisyydestä riippuen Euklidinen etäisyys, Manhattan-etäisyys, Chebyshev-etäisyys ja Octile-etäisyys (Duchoň ym. 2014; Botea ym. 2013). Manhattan-etäisyyttä käytetään pääasiassa neljän suunnan ja Octile- sekä Chebyshev-etäisyyttä kahdeksan suunnan vierekkäisyyksissä (Botea ym. 2013). Euklidista etäisyyttä voidaan käyttää tilanteessa, jossa agentti voi siirtyä seuraavaan soluun mistä kulmasta tahansa.

2.2.2 A*-algoritmin variaatiot

Reitinhakualgoritmeja toteutettiin alunperin valmiisiin ja tarkkoihin ympäristöihin, joka ei kuitenkaan ole verrattavissa reaalimaailman tilanteisiin, joissa ympäristö voi muuttua arvaamattomasti (Lawande ym. 2022). A*-algoritmia ja sen rajoitteita onkin tutkittu jo useita vuosikymmeniä, joka on mahdollistanut useiden eri variaatioiden kehittämisen. Useimmiten variaatiot keskittyvät korjaamaan yleisimpiä A*-algoritmin reitinhakuongelmia, kuten suoritustehon optimointia ja sopeutumista muuttuviin alueisiin (Stentz 1994).

Stentz (1994, 1995) esitti 90-luvulla kaksi A*-algoritmin variaatiota: D*-algoritmi (Dynamic A*) ja Focussed D*-algoritmi (Stentz 1994; Stentz ym. 1995). Uudet variaatiot pyrkivät ratkaisemaan etenkin muuttuvan ja tuntemattoman alueen ongelmat robotiikan tutkimuksessa. Alkuperäinen D*-algoritmi teki mahdolliseksi reitin korjaamisen esteen tai muutoksen tullessa reitille (Stentz 1994). Focussed D*-algoritmi tehosti alkuperäisen D*-algoritmin toimintaa ajallisesti ja näin ollen saattoi loppuun D*-algoritmin kehittämisprosessin (Stentz ym. 1995).

Vuonna 2005 Koenig ja Likhachev (2005) esittelivät D* Lite -algoritmin, joka nimestään huolimatta ei varsinaisesti perustu suoraan D*-algoritmiin, vaan A*- ja LPA*-algoritmiin (Koenig ja Likhachev 2005). D* Lite -algoritmi osoittautui yksinkertaisemmaksi ja hieman

tehokkaammaksi kuin D*- ja Focussed D*-algoritmit, jonka vuoksi sen toteuttaminen ja soveltaminen oli helpompaa (Koenig ja Likhachev 2005). (TODO: tämän kappaleen viilaus kuntoon, lainaukset, kieliasu, eteneminen jne.)

2.3 Reitinhaun haasteet

Yhden agentin staattisen ruudukkoalueen reitinhakuongelma on ratkaistavissa optimaalisesti heuristisilla reitinhakualgoritmeilla, mutta nykyään videopeleissä reitinhakuongelmat ovat monimutkaisempia ja saattavat vaatia useiden eri kriteerien täyttymisen. Suurimpia haasteita ovat esteiden järkevä väistäminen, optimaalisen reitin löytäminen ja suorituskykyvaatimusten minimointi (Abd Algfoor, Sunar ja Kolivand 2015; Cui ja Shi 2011). Näiden lisäksi reitinhakuongelmat pitävät sisällään esimerkiksi useamman agentin samanaikaista reitinhakua ja reaaliajassa muuttuvan alueen reitinhakua. Siksi videopeleissä onkin käytössä monia eri reitinhakualgoritmeja, joista osa on kehitetty toimimaan dynaamisissa ympäristöissä, osa staattisissa ympäristöissä ja osa molemmissa (Lawande ym. 2022). Reitinhakualgoritmin valinta on siis riippuvainen käyttötarkoituksesta. Nykytutkimus keskittyy pääasiassa monimutkaisten reitinhakuongelmien ratkaisemiseen ja algoritmien vertailuun erilaisissa reitinhakutilanteissa.

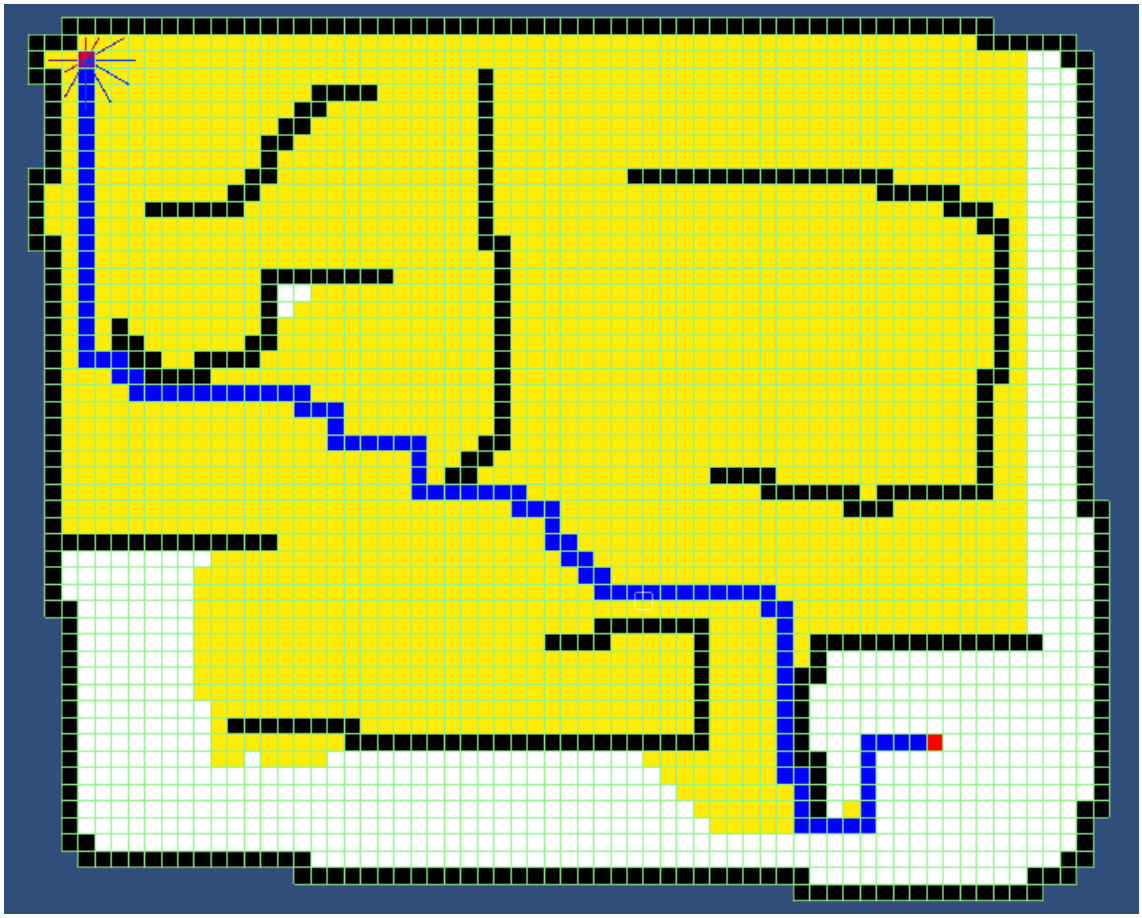
2.3.1 Suorituskyky

Reitinhakualgoritmin täytyy minimoida suoritustehon ja tallennustilan käyttö. Vaikka komponentit ovatkin kehittyneet vuosi vuodelta nopeasti, myös videopelit ovat monimutkaistuneet ja niiden laskennalliset vaatimukset kasvaneet. Reitinhakualgoritmeille varatut resurssit ovat videopeleissä rajatut, koska resursseja käytetään hyvin paljon myös graafisiin ja fysikaalisiin ominaisuuksiin (Lawande ym. 2022). Suoritustehoon liittyen etenkin muistinkäyttöä ja laskentatehoa pidetään yleisesti rajoittavina tekijöinä videopelien reitinhaussa (Botea ym. 2013). Ongelmia voidaan ratkaista erilaisilla algoritmeilla tai pelialueen esitystapaan liittyvillä ratkaisuilla (Botea ym. 2013; Cui ja Shi 2011).

Yksi suorituskykyyn liittyvistä ongelmista on reitinhakualgoritmien huono skaalautuvuus etenkin muistinkäytön suhteen. Jos agentin reitinhakuun sovelletaan A*-algoritmia suurella

1000x1000 pelialueella, muistiin joudutaan tallentamaan jopa miljoona solmua (Cui ja Shi 2011; Duchoň ym. 2014). Muistinkäyttöä havainnollistetaan kuvassa 2, jossa A*-algoritmia on käytetty esteitä sisältävän alueen reitinhakuun. Kuvan keltaiset ruudut ovat tallennettu muistiin reitinhaun aikana. Kuvasta huomaa, että A*-algoritmi käy läpi suuren määrän ylimääräisiä solmuja, koska alueella on esteitä. Ongelma moninkertaistuu, jos alueella liikkuu useita reitinhakuagentteja. Tässä tapauksessa A*-algoritmi osoittautuu riittämättömäksi ongelman ratkaisuun, joten tilanteeseen kannattaa soveltaa erilaisia menetelmiä tai algoritmien variaatioita.

TODO: tarkista vielä kuva ja sen käyttö esim. tarvitaanko? havainnollistaako tarpeeksi?



Kuvio 2. Ruudukkoalue, jossa sininen reitti on A*-algoritmin laskema optimaalisin reitti. Keltaiset ruudut ovat käsitellyt solmut, jotka on tallennettu muistiin.

TODO: lisää esimerkkejä, lisää suorituskykyongelmia

2.3.2 Dynaamisen alueen reitinhaku

Dynaamisen eli muuttuvan alueen reitinhaussa pyritään löytämään optimaalisin reitti jatkuvasti muuttuvassa alueessa (Lawande ym. 2022). Dynaamisen alueen reitinhakua esiintyy erityisesti robotiikassa, mutta myös videopeleissä. Robottien reitinhaussa ympäröivää tilaa mitataan erilaisten sensorien avulla, jonka perusteella reitinhakua suoritetaan. Videopeleissä pelialue on kokonaisuudessaan usein valmiiksi reitinhakuagentin tiedossa eikä erillisiä sensoreita käytetä pelialueen havainnoimiseen, ellei haluta vähentää epäreiluutta ja matkia ihmismäistä käyttäytymistä. (Rahmani ja Pelechano 2022).

Videopeleissä pelialue voi muuttua toisten pelaajien, ei-pelaaja-hahmojen tai muuten vain liikkuvien esteiden takia. Esimerkiksi kilpa-ajoneuvopeleissä pelaaja voi kisata ei-pelaaja-hahmon kanssa, joka joutuu väistelemään sekä pelaajan ajoneuvoa että muita alueen liikkuvia esteitä (Sazaki, Primanita ja Syahroyni 2017). Jokainen alueen muutos rikkoo reitinhakugraafin rakenteen, jolloin graafi ja valittu reitti täytyy korjata. Graafin luominen voi olla isoissa pelialueissa kallis operaatio, joten jatkuva uudelleenluominen ja uuden reitin etsiminen ei välttämättä ole vaihtoehto. Nykyään videopelien reitinhaun tutkimuksissa painotetaankin dynaamisia reitinhakualgoritmeja.

Graafin jatkuva päivittäminen on tuonut erilaisia ideoita tutkimuksissa. Luvussa 2.2.2 mainittu D*-algoritmi on kehitetty juuri tuntemattoman ja muuttuvan alueen reitinhakuun.

TODO: artikkeleita dynaamisesta hausta ja sen ongelmista

2.3.3 Moniagenttireitinhaku

Moniagenttireitinhaussa alueella on useampi kuin yksi agentti ja jokaisella niistä on oma aloitus- ja lopetuspisteensä graafissa. Jokaisella aika-askeleella agentti voi joko liikkua toiseen solmuun tai pysyä paikallaan nykyisessä solmussaan. (Sharon ym. 2015; Stern ym. 2019). Tarkoituksena on samaan aikaan ratkaista moniagenttireitinhakuongelma ja minimoida reitinhaun kustannusfunktio (engl. cost function). Yleinen kustannusfunktio on kustannusten summa (engl. sum-of-costs), joka on agenttien aika-askelten summa kun ne saapuvat kohteisiinsa. Kustannusfunktiona voi olla myös kokonaiskesto (engl. makespan), joka minimoi ajan kunnes viimeinen agentti on saapunut kohdesolmuun tai polttoaine (engl. fuel), joka

minimoi agenttien kulkeman matkan. Erityisenä huomiona tässä kustannusfunktiossa on se, että paikallaan odottaminen ei nosta kustannuksia. (Sharon ym. 2015).

Moniagenttireitinhaun rajoitteet voivat vaihdella tutkimusalasta riippuen, kuten esimerkiksi saavatko agentit kulkea samaa reittiä pitkin seuraten toisiaan, mutta yleensä perusrajoitteet ovat samat (Sharon ym. 2015; Stern ym. 2019). Perinteisesti agenttien täytyy liikkua lopetuspisteeseen törmäämättä toisiinsa matkalla ts. yhdellä solmulla ei saa olla samanaikaisesti enempää kuin yksi agentti. Kaksi agenttia eivät myöskään saa kulkea saman kaaren kautta samalla aika-askeleella. Moniagenttireitinhakuongelman sovelluksia esiintyy videopelien lisäksi esimerkiksi robotiikassa, ilmailussa, liikennesuunnittelussa ja itseohjautuvissa ajoneuvoissa, joten se on saanut viime vuosina paljon huomiota tutkimuksissa ja akateemisissa yhteisöissä. (Sharon ym. 2015; Stern ym. 2019).

Moniagenttireitinhakuongelmaan on sekä optimaalisia että epäoptimaalisia ratkaisuja. Optimaaliset ratkaisut ovat luonteeltaan NP-kovia, koska agenttien lukumäärä kasvattaa ongelman tila-avaruutta eksponentiaalisesti. A*-algoritmi on esimerkiksi optimaalinen ratkaisu, mutta moniagenttireitinhaussa sen suoritusaika voi olla hyvin pitkä ja muistinkäyttö liian suurta. (Sharon ym. 2015). Tämän vuoksi ongelmissa, joissa agentien lukumäärä on suuri, käytetään usein epäoptimaalisia ratkaisuja.

TODO: moniagenttireitinhaku oikea termi?

3 Koneoppiminen

Koneoppiminen on tällä hetkellä yksi teknisten tutkimusalojen suosituimmista aihealueista. Sen ydin rakentuu kysymykselle, pystyykö tietokone jäljittelemään ihmismielen oppimisprosessia ja täten oppia automaattisesti kokemuksen kautta. (Das ja Behera 2017; Jordan ja Mitchell 2015). Automaattisella oppimisella pyritään vähentämään manuaalista, tapaus tapauksen perään ohjelmoimista. Sen sijaan konetta opetetaan syöte-tuloste -parien avulla. Vuosikymmenten aikana koneoppimisen tutkimus on edistynyt hyvin paljon eikä loppua ole toistaiseksi näkymässä. Uudet sovellukset ja algoritmit, laskentatehon kasvu ja big data eli hyvin suuret, keskittyneet datamäärät ovat tuoneet tarpeen sekä teorian tutkimukselle että kehittyneille käytännön ratkaisuille.

3.1 Koneoppimisen perusteet

Koneoppiminen on tieteenalana yhdistelmä tietotekniikkaa ja tilastotieteitä. Tietotekniikka ja tietokoneet mahdollistavat ongelmanratkaisun. Suurista datajoukoista oppiminen, erilaisten ennusteiden tekeminen ja päätöksenteko vaativat sen sijaan tilastotieteellisiä menetelmiä. (Das ja Behera 2017; Jordan ja Mitchell 2015). Myös neurotieteiden ja psykologian roolit ovat kasvamassa koneoppimisen tutkimuksessa. Esimerkiksi ihmisaivojen ja sitä kautta oppimisprosessin tutkiminen ja hyödyntäminen koneoppimisessa ovat tulevaisuudessa merkittäviä tutkimuksen kohteita. (Das ja Behera 2017). Koneoppimisen sovellukset ovat nykyään hyvin laaja-alaiset. Tunnettuja esimerkkejä ovat konenäkö, puheentunnistus ja robotiikka. Konenäköä hyödynnetään etenkin terveydenhoidon alalla anomalioiden tunnistamiseen kuvissa. Mainonnassa koneoppimista käytetään personoitujen suositusten luomiseen ja markkinoinnissa erilaisissa ennusteissa.

Tietoteknisten laitteiden suosio ja sitä kautta datan räjähdysmäinen kasvu ovat olleet suuressa roolissa koneoppimisen hyödyntämisessä. Kyseisiä suuria datamassoja eli big dataa on mahdotonta tarkastella manuaalisesti, joten niiden käsittelyyn on otettu käyttöön koneoppimisen menetelmiä. Koneoppimisalgoritmit pystyvät muokkaamaan palveluita vastaamaan jokaisen henkilökohtaisiin tarpeisiin personoidun datan avulla. Esimerkiksi mainoksia voidaan koh-

dentaa tietyille ryhmille ja vanhoja potilastietoja voidaan hyödyntää hoitotyypin valitsemiseen tietyille potilaille. (Jordan ja Mitchell 2015). Personoitu data tuo tosin tietoturvallisia haasteita. Henkilödatasta on pyrittävä tekemään anonyymia, jottei sitä pysty yhdistämään suoraan henkilöihin. Samalla kuitenkin data voi olla niin tarkkaa, että jokaisella sanotaan olevan oma digitaalinen sormenjälki suuressa datamassassa. Myös suorituskykyvaatimukset ovat nousseet datan kasvun myötä, joten algoritmit täytyy kehittää mukautuviksi (Jordan ja Mitchell 2015).

Koneoppimisella yritetään perinteisesti ratkaista luokitteluongelmia, joissa datajoukosta voidaan päätellä, kuuluuko kyseinen asia tiettyyn luokkaan vai ei (Jordan ja Mitchell 2015). Esimerkiksi sähköposti voidaan luokitella roskapostiksi tai aidoksi sähköpostiksi riippuen mitkä ovat luokittelun vaihtoehdot. Koneoppimisen avulla kasvatetaan luokittelun tarkkuutta, joka on kyseisen luokitteluongelman tärkein mittari. Esimerkkidata, jolla konetta opetetaan, voi koostua kokoelmasta erilaisia sähköposteja, jotka on valmiiksi luokiteltu roskapostiksi tai aidoksi sähköpostiksi. Koneoppimisen avulla kone opetetaan tunnistamaan malleja ja kuvioita datassa, jolloin se pystyy myöhemmin tunnistamaan samoja kuvioita oikeissa tilanteissa. (Jordan ja Mitchell 2015).

Koneoppiminen voidaan jakaa ohjattuun oppimiseen, ohjaamattomaan oppimiseen ja vahvistusoppimiseen. Ohjatussa oppimisessa oppiminen tapahtuu luokitellun aineiston perusteella. Se havainnollistaa lineaariapproksimaatiota ja siihen liittyviä ongelmia. Opetusaineisto tuodaan (x, y) pareina, jossa x on syöte, esimerkiksi kuva, ja y on tulos, esimerkiksi lintu. Oppijan tavoitteena on tuottaa ennuste y syötteen x perusteella.

TODO: tästä eteenpäin eri oppimisen osa-alueet ja loput alaotsikot.

- Ohjattu oppiminen on koneoppimisen osa-alue, jossa oppiminen tapahtuu luokitellun aineiston perusteella. Ohjattu oppiminen havainnollistaa lineaariapproksimaatiota ja siihen liittyviä ongelmia. Opetusaineisto tuodaan (x, y) pareina, jossa x on syöte, esimerkiksi kuva, ja y on tulos, esimerkiksi lintu. Oppijan tavoitteena on tuottaa ennuste y syötteen x perusteella.

- Ohjaamaton oppiminen mahdollistaa koneoppimisen ilman ihmisen apua tai väliintuloa. Ohjaamattomassa oppimisessa pyritään tunnistamaan datan rakenteet ja mallit ilman datan valmista luokittelua.

- Vahvistusoppiminen sijoittuu ohjatun ja ohjaamattoman oppimisen väliin. Harjoitusdata on vain osittain luokiteltua, ja suurin osa oppimisesta tapahtuu kokeilemalla. Tietylle syötteelle x ei siis anneta valmiiksi oikeaa tulosta y, vaan agentti saa jokaisesta toiminnasta positiivisen tai negatiivisen palautteen tai palkkion. Agentin tarkoituksena on maksimoida palkkioiden summa tehtävää suorittaessa. Vahvistusoppimisesta selitetään luvussa 3.3 lisää.

3.2 Neuroverkot ja syväoppiminen

-

3.3 Vahvistusoppiminen ja syvä vahvistusoppiminen

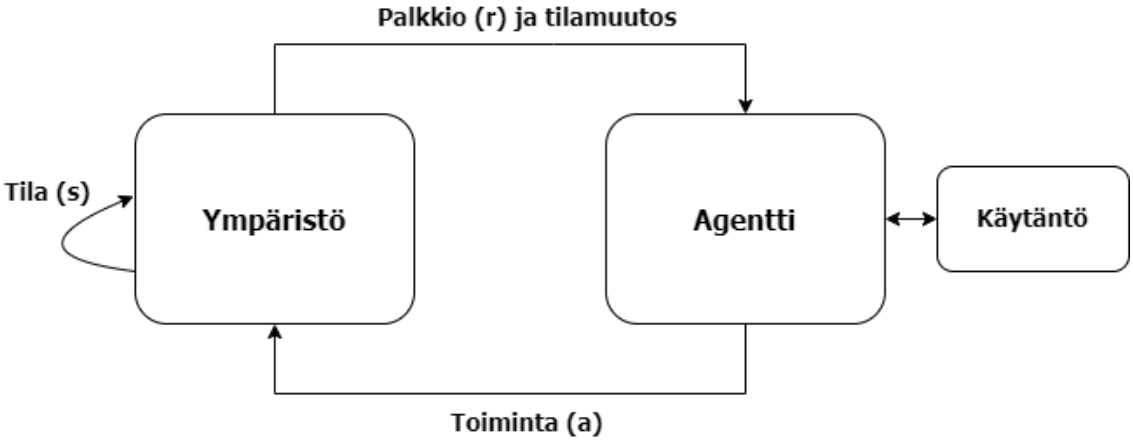
Vahvistusoppiminen on koneoppimisen osa-alue, jonka tavoitteena on opettaa ja ohjata agentin käyttäytymistä yrityksen ja erehdyksen kautta. Agentti on jatkuvassa vuorovaikutuksessa ympäristönsä kanssa ja tarkkailee samalla toimintojensa vaikutusta ympäristössä. Agentin oppimisprosessia ja käyttäytymistä ohjataan palkkioilla. (Arulkumaran ym. 2017)

Vahvistusoppimisen malli voidaan kuvata tilojen S, toimintojen A ja palkkiosignaalien r avulla. Jokaisella aika-askeleella t agentti suorittaa toiminnan a, jolloin ympäristön tila s muuttuu ja muutos viestitään agentille palkkiosignaalin r kautta. Palkkiosignaalin lisäksi ympäristö viestii agentille tilamuutoksesta, jolloin agentti saa tiedon uudesta tilasta $s+1$. Palkkion ja uuden tilan perusteella agentti valitsee jälleen seuraavan toimintonsa. Agentin lopullisena tavoitteena on saavuttaa paras mahdollinen käytäntö (engl. policy), joka maksimoi saadun palkkion määrän. (Arulkumaran ym. 2017). Kuva 3 havainnollistaa vahvistusoppimisen toimintaa. TODO: kuvan käyttäminen järkevämmin?

TODO: rise of drl eteenpäin, actor-critic menetelmästä asiaa?

3.4 Soft-Actor Critic -algoritmi

Soft Actor-Critic on Haarnojan ym. kehittämä algoritmi, joka perustuu syvään vahvistusoppimiseen (Haarnoja ym. 2018).



Kuvio 3. Vahvistusoppimisympäristön oppimissilmukka yhdellä aika-askeleella.

4 Unity

Unity on Unity Technologiesin kehittämä pelinkehitysalusta, joka sisältää oman renderöinti- ja fysiikkamoottorin sekä Unity Editor -nimisen graafisen käyttöliittymän (Juliani ym. 2018). Unityllä on mahdollista kehittää perinteisten 3D- ja 2D-pelien lisäksi myös esimerkiksi VR-pelejä tietokoneille, mobiililaitteille ja pelikonsoleille. Unitystä onkin vuosien mittaan tullut yksi tunnetuimmista pelinkehitysalustoista, jonka parissa työskentelee kuukausittain jopa 1.5 miljoonaa aktiivista käyttäjää ("Unity" 2022).

Viime vuosina Unityä on käytetty simulointialustana tekoälytutkimuksen parissa. Unity mahdollistaa lähes mielivaltaisten tilanteiden ja ympäristöjen simuloinnin 2D ruudukkokartoista monimutkaisiin pulmanratkaisutehtäviin, joka on sen suurimpia vahvuuksia simulointialustana. Kehitystyö ja prototypointi ovat Unityllä myös erityisen nopeaa. (Juliani ym. 2018).

4.1 Unityn hierarkia

Tässä luvussa käsitellään Unityn hierarkiaa. Aliluvuissa käydään läpi hierarkian osat ylimmästä lähtien.

4.1.1 Unity-projekti

Unityn hierarkian ylin osa on projekti, jonka luomisesta kehitystyö aina alkaa. Unityssä on mahdollista luoda projekti valmiista pohjista, joita ovat esimerkiksi 2D-, 3D- ja VR-pohjat. Pohjien avulla projekteihin saa lisättyä suoraan suositeltavat, parhaita käytäntöjä mukailevat asetukset.

Unity-projekteja voidaan hallinnoida ja avata erillisellä Unity Hub -sovelluksella. Unity Hub kertoo muun muassa mitä Unityn versiota projekti tukee. Projekteja voi tarvittaessa siirtää (engl. migrate) toimimaan uusimmilla Unityn versioilla, mutta siirto voi aiheuttaa toiminnallisuuksien muutoksia tai virheitä projektissa.

4.1.2 Näkymät

Projektista seuraavana hierarkiassa ovat näkymät (engl. scene). Näkymät toimivat työskentelyalustoina projektissa. Projekti sisältää aina yhden tai useamman näkymän, koska ilman niitä mitään ei pysty luomaan. Tavallisesti yksittäinen näkymä kuvaa aina yhtä kenttää, tasoa tai aluetta pelissä, ja siirryttäessä toiselle alueelle Unity pystyy lataamaan ajon aikana uuden skenen. Näkymän lataaminen voi tosin viedä aikaa, joten lataus peitetään useimmiten latausruuduilla, jotka voivat myös olla omia, yksinkertaisia näkymiä. Yksinkertaisimmillaan peli voi kuitenkin sisältää vain yhden näkymän, joka muokkautuu ja jota muokataan ajon aikana.

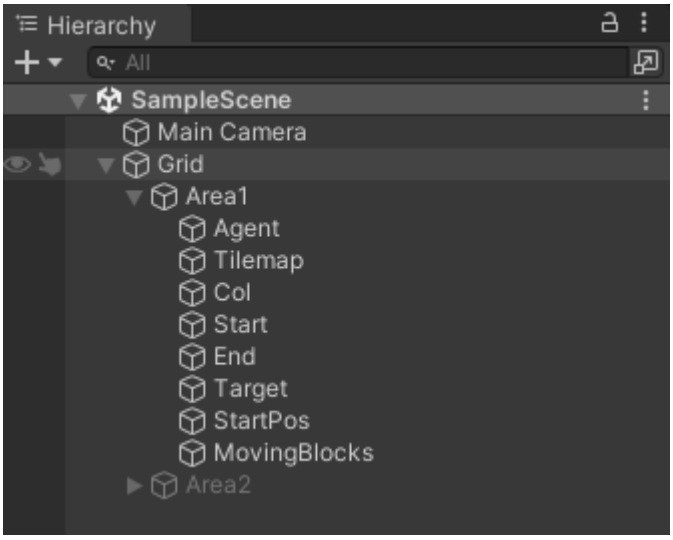
Projektin luonnin jälkeen Unity lisää siihen automaattisesti aloitusnäkymän, joka sisältää kamera- ja valonlähde-peliobjektin. Tätä näkymää voi lähteä muokkaamaan lisäämällä siihen erilaisia peliobjekteja, esimerkiksi maata ja erilaisia geometrisia muotoja.

4.1.3 Peliobjektit ja Prefabit

Peliobjektit (engl. GameObject) ovat tärkein osa Unityn pelinkehitysprosessia, koska kaikki peliin luotavat objektit ovat taustaltaan peliobjekteja. Peliobjektit eivät itsessään tee mitään tai näytä miltään, vaan ne toimivat säiliöinä komponenteille. Peliobjekteja voidaan järjestellä vanhempi-lapsi -periaatteella. Lapsiobjektit liikkuvat vanhemman mukana, jolloin niitä ei tarvitse liikutella näkymässä erikseen. Kuva 4 havainnollistaa esimerkkinäkymän peliobjekteja ja niiden vanhempi-lapsi -suhteita.

Prefabit ovat peliobjektien valmiita malleja, joita luodaan peliobjektien tavoin. Prefabit vähentävät toistuvaa työtä peliobjektien luomisen yhteydessä. Prefabeille voidaan lisätä komponentteja ja lapsiobjekteja kuten peliobjekteille. Kun prefabi lisätään näkymään, sen komponentteja ja arvoja voidaan muuttaa tarvittaessa ilman, että alkuperäisen prefabin arvot ja komponentit muuttuvat.

Peliobjekteja voi lisätä "GameObject-valikosta Unityn ylävalikosta joko tyhjinä objekteina tai valmiina kokonaisuuksina. Valmiita peliobjekteja ovat esimerkiksi erilaiset valonlähteet tai 3D-objektit, ja ne sisältävät automaattisesti tarvittavat komponentit.



Kuvio 4. Lista peliobjekteista avoimessa näkymässä. Sisäkkäiset peliobjektit ovat lapsiobjekteja.

4.1.4 Komponentit

Komponentit antavat peliobjekteille ominaisuuksia ja toiminnallisuuksia kuten muodon, värin tai fysiikan. Komponentteja voi olla rajattomasti, mutta peliobjektilla on luonnin jälkeen vähintään Transform-komponentti, joka määrittää peliobjektin sijainnin, suunnan ja skaalan. Transform-komponenttia ei voi poistaa peliobjektilta. Esimerkiksi valmis 3D-objekti pallo (sphere) saa automaattisesti Mesh Filter-, Mesh Renderer- ja Sphere Collider-komponentit. Kaksi ensimmäistä komponenttia keskittyvät pallon graafisiin ominaisuuksiin ja Sphere Collider fyysisiin törmäysominaisuuksiin. Sphere Collider-komponentti asettuu automaattisesti pallon graafisen ulkomuodon kokoiseksi.

Usein komponenteilla on erilaisia arvoja, kuten koko tai väri, joita voi muokata käyttöliittymäelementtien avulla. Komponentit voivat sisältää myös viittauksia muihin peliobjekteihin, tiedostoihin tai assetteihin(käännös?). Esimerkiksi Sprite Renderer -komponenttiin voidaan lisätä viittaus kuvatiedostoon, jolloin Unity renderöi peliobjektin kohdalle lisätyn kuvan. Kuvassa 5 on lista peliobjektin komponenteista ja

4.1.5 Skriptit

Ohjelmoinnin merkitys Unityn käytössä tulee skripteistä. Skriptit ovat ohjelmakooditiedostoja, joita voi lisätä peliobjektiin komponentin tavoin, jos valmiit komponentit eivät riitä toiminnallisuuksiltaan. Skripteissä voi esimerkiksi määritellä ominaisuuksia ja arvoja, joita voi muokata komponenttilistauksessa tai ajon aikana. Unity tukee tällä hetkellä vain C-# -ohjelmointikieltä, mutta ennen myös Javascriptiin pohjautuvaa UnityScript-ohjelmointikieltä.

Unity tarjoaa skripteihin MonoBehaviour-pohjaluokan, joka mahdollistaa pelinkehityksen tärkeimmät osat eli Start()-aloitusfunktion ja Update()-päivitysfunktion. Start()-funktio ajetaan ennen yhtäkään päivitysfunktiota, joten siinä voidaan määrittää ja alustaa tarvittavat alkuarvot. Update()-funktio ajetaan joka ruudunpäivityskerralla, joten siihen sijoitetaan usein pelilogiikka ja mahdollisesti fyysiset toiminnallisuudet.

Skriptien avulla voidaan tehdä lähes kaikki samat asiat kuin Unity Editorissa. Peliobjekteja voidaan etsiä tagien tai nimien kautta ja niille voidaan lisätä ja poistaa komponentteja ajon aikana. Käyttämättömät peliobjektit voidaan tuhota tai asettaa epäaktiivisiksi jos niitä ei tarvita.

4.2 Machine Learning Agents

Machine Learning Agents on Unitylle kehitetty ilmainen koneoppimispaketti, joka mahdollistaa Unity Editorilla luotujen simulaatioympäristöjen ja Python APIn välisen vuorovaikutuksen. ML-Agents SDK (Software Development Kit) tarjoaa kaikki toiminnallisuudet ja skriptit toimivan koneoppimisympäristön luomiseen. (Juliani ym. 2018). Kuva 6 havainnollistaa Unity Editorilla luodun yksinkertaisen ML-Agents koneoppimisympäristön toimintaa. TODO: "unity editorilla luodun"vai pelkästään "ML-agentsin toimintaa/rakennetta"

4.2.1 ML-Agents SDK

ML-Agents SDK sisältää kolme ydinosaa: sensorit, agentit ja akatemia. Agentti-komponentti voidaan lisätä suoraan Unityn peliobjektille, jolloin se pystyy keräämään havaintoja, suorittamaan toimintoja ja vastaanottamaan palkkioita. Sensorit mahdollistavat havaintojen kerää-

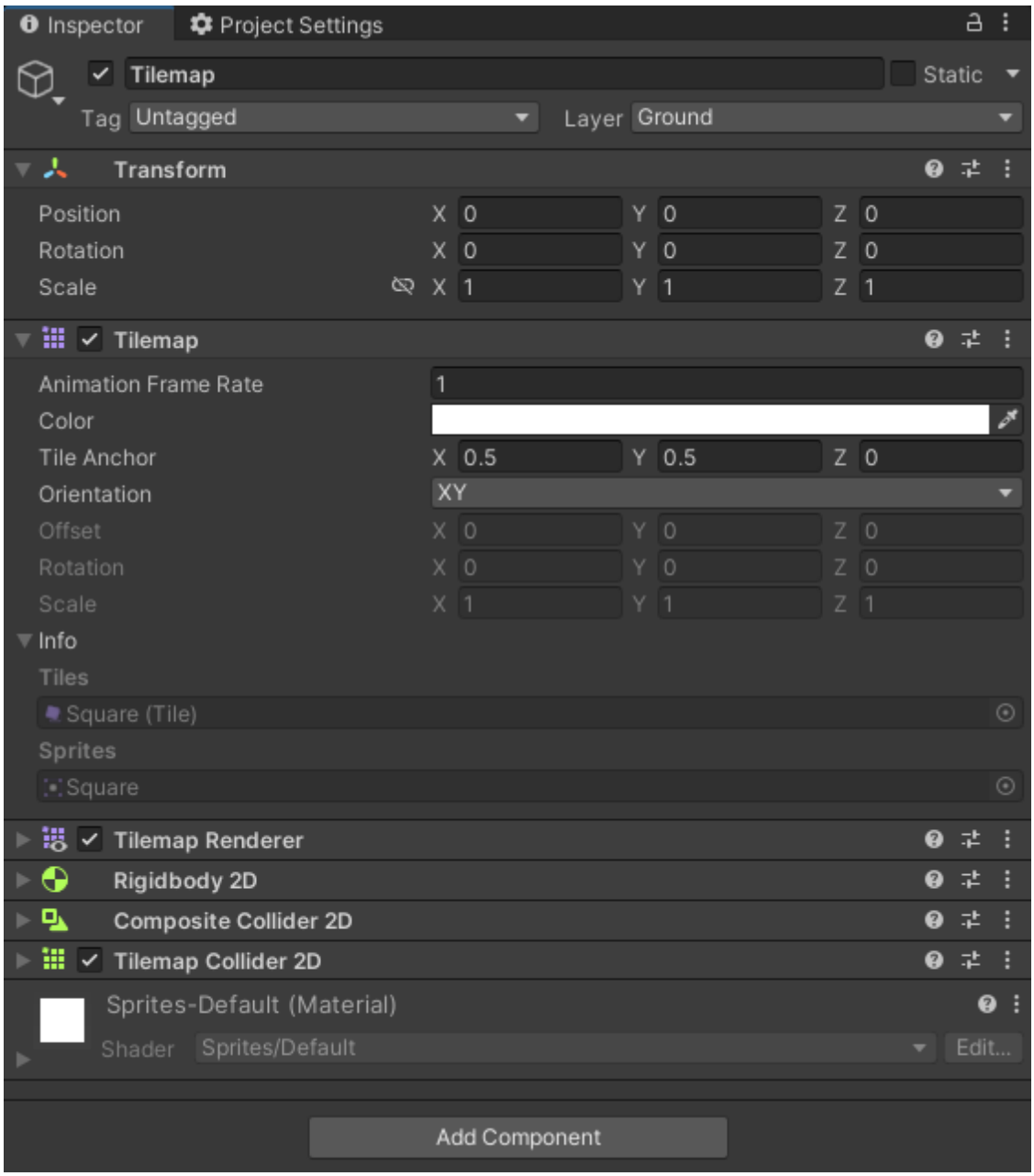
misen eri tavoin. Akatemia ylläpitää tietoa simulaation askelmäärästä ja ympäristön parametreista sekä ohjaa agenttien toimintaa.

Agentin käytäntö määritellään Behavior Name -nimikkeen avulla. Eri agenteilla voi olla sama käytäntö, jolloin agentit käyttävät kyseistä käytäntöä päätöksentekoon ja jakavat harjoitteludatan keskenään. Myös useiden erilaisten agenttien toiminta voidaan mahdollistaa erinimisillä käytännöillä.

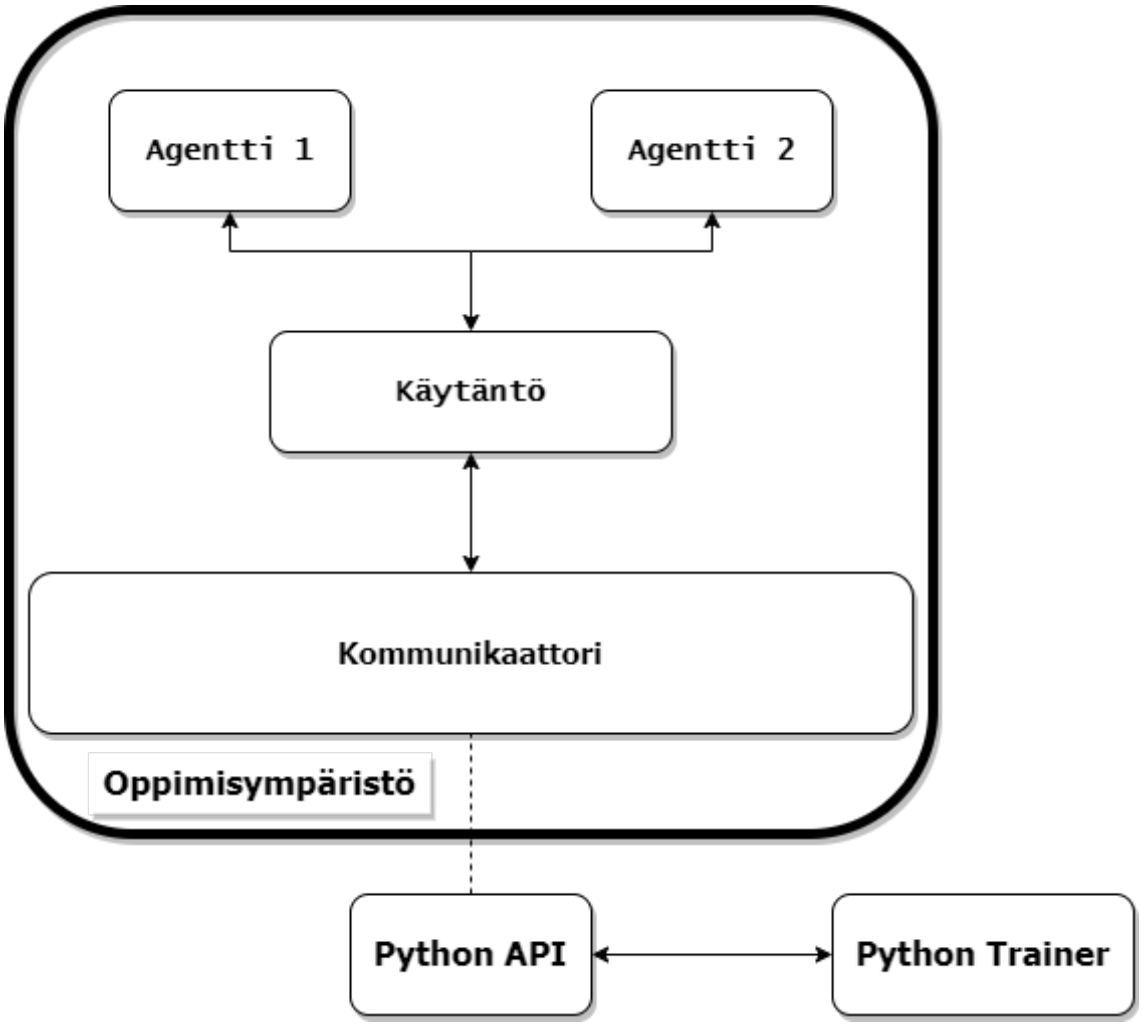
4.2.2 Python API ja PyTorch

Python APIa käytetään Unityllä tehdyn simulaatioympäristön ja koneoppimissilmukan käsittelyyn. APIn avulla tekijän ei tarvitse itse olla suoraan yhteydessä Pythonin koneoppimiskouluttajaan, vaan API tarjoaa helppokäyttöiset, valmiit menetelmät koneoppimissilmukan luomiseen. Tarkemmin APIsta ja sen toiminnasta voi lukea dokumentaatiosta (tähän linkki sivun alalaitaan tulevasta docsista?).

PyTorch on avoimen lähdekoodin koneoppimiskehys, johon pohjautuen Unity ML-agents -paketin koneoppimiseen liittyvät toteutukset on tehty (Unity ML agents docs). PyTorch sisältää kaikki syvän koneoppimisen perusosat datan kanssa työskentelystä ja koneoppimismallin luomisesta mallin parametrien optimointiin ja oppimismallien tallentamiseen (PyTorch docs).



Kuvio 5. Lista peliobjektin komponenteista.



Kuvio 6. Lohkokaavio Unity ML-Agents toiminnasta.

5 Tutkimuksen empiirinen osuus

Tässä kappaleessa käsitellään tutkimuksen empiiristä osuutta. Tutkimus toteutettiin empiirisenä vertailevana tutkimuksena. Vertailun kohteena olivat Unityn ML-agentin suorittama reitinhaku ja heuristiseen A*-algoritmiin perustuva reitinhaku. Kappaleessa 5.1 kuvaillaan tutkimusta yleisellä tasolla ja esitellään tutkimuksessa käytetyt työkalut. Kappaleessa 5.2 käydään läpi simulaatioympäristöt ja koneoppimiseen liittyvät konfiguraatiot. Lopuksi kappaleessa 5.3 käsitellään tulosten mittaamista ja käytettyjä suureita.

5.1 Tutkimuksen kuvaus

Tutkimuksen simulaatioalustana käytettiin Unityä, koska sen käyttö oli ennestään tuttua ja se soveltuu hyvin erilaisten tilanteiden simuloimiseen. Unity mahdollisti valmiit menetelmät ruudukkoalueen toteutukseen, jolloin ympäristön luomiseen ei mennyt liikaa aikaa. Tutkimusta varten luotiin valmis 2D-projekti ja projektiin lisättiin Unity Package Managerin avulla ML-Agents -paketti, jotta tarvittavat toiminnallisuudet saatiin käyttöön koneoppimista varten. Unityllä luotiin yksinkertaisia ruudukkoalueita, jonne sijoitettiin esteruutuja, kävelykelpoisia ruutuja ja alku- ja loppuruudut sekä joissain alueissa liikkuvia esteitä. Reitinhaun vertailua varten luotiin A*-algoritmi ja erikseen opetettiin koneoppimisagentti. Koneoppimisagentti käyttää syvään vahvistettuun oppimiseen perustuvaa SAC-algoritmia.

A*-algoritmia täytyi muokata tutkimukseen sopivaksi. Pelialueet ovat dynaamisia eli muuttuvat reaaliaikaisesti, joten perinteinen A*-algoritmi ei pysty reagoimaan muutoksiin välittömästi. Algoritmia muokattiin tutkimukseen siten, että esteen ilmestyessä laskelmoidulle reitille A*-algoritmi ajetaan uudestaan, mutta lähtöpisteeksi asetetaan agentin uusi sijainti. Uusi sijainti on agentin siihen asti etenemä matka, jolloin seuraava reitinhakuiteraatio vaatii lyhyemmän matkan. Jokainen iteraatio on myös suurimmassa osassa tapauksista kevyempiä muistinkäytön kannalta.

5.2 Tutkimusasetelma / Konfiguraatio

Koneoppimisagentin toiminta riippuu vahvasti sen asetuksista.

- AddObservation()-funktion avulla agentti kerää tietoa omasta tilastaan ympäristössä. Tässä tutkimuksessa agentin täytyy kerätä tietoa sen omasta sijainnista (x ja y), mahdollisesti etäisyydestä maaliin (toimisiko esim. laskea manhattan-etäisyys?) ja maalin sijainnista (x ja y). Suositeltavaa kuitenkin on, että agentti keräisi tietoa vain näkemistään asioista eli tässä tapauksessa vain omasta sijainnistaan. Näiden havaintojen lisäksi käytetään RayPerceptionSensor2D-komponenttia, joka kerää ympäröivästä alueesta tietoa säteiden avulla. Tässä projektissa agentista lähetetään 12 sädettä, jotka yltävät vain yhden ruudun verran eteenpäin. Säteet pystyvät tunnistamaan vastaantulevia objekteja, joita ovat tässä tapauksessa esteet ja maaliruutu. Näiden havaintojen avulla agentti voi tehostaa päätöksentekoaan.

- Actions, toiminnat, yksinkertaisen pelialueen takia käytetään Discrete Actions ja viittä eri vaihtoehtoa (ylös, alas, oikealle, vasemmalle, paikallaan). Jos pelialue olisi monimutkaisempi, voidaan käyttää Continuous Actions (suunta + nopeus).

- Rewards, palkinnot, agentti saa positiivisen palkkion kun saapuu maaliin (+1). Agentti saa negatiivisen palkkion jokaisen toiminnan (action) jälkeen (-0.01). Jos agentti törmäilee seiniin, eli yrittää liikkua suuntaan jossa on este, agentille annetaan negatiivinen palkkio (-0.001). Palkkiot määritellään skriptitiedostossa ja ovat helposti muutettavissa.

- Agentin opettaminen aloitetaan komentoriviltä komennolla ml-agents "BehaviorName"... (myös jotain muuta, ml-agents -komento hyödyntää siis Python APIa). Komentorivi näyttää käyttäjän asettamien opetusaskelten välein mm. agentin keskiarvopalkkion askelten aikana. Tässä tutkimuksessa asetettiin väliksi 10000, tensorboard päivittyy myös 10000 välein ts. piirtää kuvaajan pisteen?

5.3 Tulosten mittaaminen

- Reitinhakutehtävästä otetaan talteen aika (sekuntia) ja onnistuiko reitinhaku vai ei (onnistui/epäonnistui).

6 Tulokset ja johtopäätökset

7 Yhteenveto

Lähteet

Abd Algfoor, Zeyad, Mohd Shahrizal Sunar ja Hoshang Kolivand. 2015. “A comprehensive study on pathfinding techniques for robotics and video games”. *International Journal of Computer Games Technology* 2015.

Arulkumaran, Kai, Marc Peter Deisenroth, Miles Brundage ja Anil Anthony Bharath. 2017. “A brief survey of deep reinforcement learning”. *arXiv preprint arXiv:1708.05866.*

Botea, Adi, Bruno Bouzy, Michael Buro, Christian Bauckhage ja Dana Nau. 2013. “Pathfinding in games”. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Cui, Xiao, ja Hao Shi. 2011. “A*-based pathfinding in modern computer games”. *International Journal of Computer Science and Network Security* 11 (1): 125–130.

Das, Kajaree, ja Rabi Narayan Behera. 2017. “A survey on machine learning: concept, algorithms and applications”. *International Journal of Innovative Research in Computer and Communication Engineering* 5 (2): 1301–1309.

Demyen, Douglas, ja Michael Buro. 2006. “Efficient triangulation-based pathfinding”. Teoksessa *Aaai,* 6:942–947.

Dijkstra, Edsger W, ym. 1959. “A note on two problems in connexion with graphs”. *Numerische mathematik* 1 (1): 269–271.

Duchoň, František, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico ja Ladislav Jurišica. 2014. “Path planning with modified a star algorithm for a mobile robot”. *Procedia Engineering* 96:59–69.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel ja Sergey Levine. 2018. “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor”. Teoksessa *International conference on machine learning,* 1861–1870. PMLR.

Hart, Peter E, Nils J Nilsson ja Bertram Raphael. 1968. “A formal basis for the heuristic determination of minimum cost paths”. *IEEE transactions on Systems Science and Cybernetics* 4 (2): 100–107.

Jordan, Michael I, ja Tom M Mitchell. 2015. “Machine learning: Trends, perspectives, and prospects”. *Science* 349 (6245): 255–260.

Juliani, Arthur, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, Yuan Gao, Hunter Henry, Marwan Mattar ym. 2018. “Unity: A general platform for intelligent agents”. *arXiv preprint arXiv:1809.02627.*

Karur, Karthik, Nitin Sharma, Chinmay Dharmatti ja Joshua E Siegel. 2021. “A survey of path planning algorithms for mobile robots”. *Vehicles* 3 (3): 448–468.

Koenig, Sven, ja Maxim Likhachev. 2005. “Fast replanning for navigation in unknown terrain”. *IEEE Transactions on Robotics* 21 (3): 354–363.

Lawande, Sharmad Rajnish, Graceline Jasmine, Jani Anbarasi ja Lila Iznita Izhar. 2022. “A Systematic Review and Analysis of Intelligence-Based Pathfinding Algorithms in the Field of Video Games”. *Applied Sciences* 12 (11): 5499.

Rahmani, Vahid, ja Nuria Pelechano. 2022. “Towards a human-like approach to path finding”. *Computers & Graphics* 102:164–174.

Sazaki, Yoppy, Anggina Primanita ja Muhammad Syahroyni. 2017. “Pathfinding car racing game using dynamic pathfinding algorithm and algorithm A”. Teoksessa *2017 3rd International Conference on Wireless and Telematics (ICWT),* 164–169. IEEE.

Sharon, Guni, Roni Stern, Ariel Felner ja Nathan R Sturtevant. 2015. “Conflict-based search for optimal multi-agent pathfinding”. *Artificial Intelligence* 219:40–66.

Stentz, Anthony. 1994. “Optimal and efficient path planning for partially-known environments”. Teoksessa *Intelligent unmanned ground vehicles,* 203–220. Springer.

Stentz, Anthony, ym. 1995. “The focussed dˆ* algorithm for real-time replanning”. Teoksessa *IJCAI,* 95:1652–1659.

Stern, Roni, Nathan R Sturtevant, Ariel Felner, Sven Koenig, Hang Ma, Thayne T Walker, Jiaoyang Li, Dor Atzmon, Liron Cohen, TK Satish Kumar ym. 2019. “Multi-agent pathfinding: Definitions, variants, and benchmarks”. Teoksessa *Twelfth Annual Symposium on Combinatorial Search.*

“Unity”. 2022. Viitattu 4. huhtikuuta 2022. https://unity.com/.

“Unity Docs”. 2022. Viitattu 7. syyskuuta 2022. https://docs.unity3d.com/2023.1/Documentation/Manual/nav-InnerWorkings.html.

8 Liitteet

- Kuvat tai mallinnokset pelialueesta.

- Tensorboardin kuvaajat mm. agentin palkkioiden kehityksestä.