Agent based Koen van Heertum 26/11/2020

1. NetLogo kan makkelijk veel 'turtles' (andere benaming voor agents) aanmaken. Deze turtles kunnen namelijk op hun 'surroundings' reageren. NetLogo heeft een duidelijk verschil tussen frontend, backend en info (documentatie). Hierdoor is het programma makkelijk te gebruiken. Het gebruikt dan ook weinig regels code om een resultaat te tonen.
   Nadeel is dat NetLogo oud is. Bovendien is het niet erg uitbreidbaar, wat betekent dat het niet kan interacten met andere programma's en dat het dus ongeschikt zal zijn in grootschalige projecten.
   Bovendien zal het waarschijnlijk niet gebruikt worden door professionals aangezien het niet altijd een goede keuze is. NetLogo is gemaakt om bepaalde tasks goed te kunnen runnen, en als je project buiten die scope valt dan zal NetLogo daar moeite mee hebben op het gebied van performance.

   Mesa maakt ook gebruik van agents. De agents kunnen namelijk informatie opvragen van het model. Het werkt in Python samen met JavaScript om een visualisation te maken in je webbrowser. Hierdoor kun je mooie resultaten krijgen die van redelijk hoge qualiteit zijn. Bovendien is Mesa erg uitbreidbaar aangezien het deels via JavaScript werkt.
   Het nadeel is dat Mesa niet simpel is op te zetten. Het vereist aardig wat kennis van Python en JS, en het vereist ook aardig wat regels code voordat je een Agent Based model op hebt gezet.

   Unity KAN gebruik maken van agents. Het ding van Unity is namelijk dat het eigenlijk puur gemaakt is om videogames in te maken. Hiervoor is het programma behoorlijk goed. Bovendien bied Unity ontzettend veel opties aan. Eigenlijk te veel zelfs. Hierdoor is het wel mogelijk om agent based simulation in Unity te doen, ondanks dat het daar niet voor gemaakt was.
   Nadeel komt dan ook op hetzelfde neer. Unity is gemaakt en ge-optimized voor games maken. Omdat het programma veel opties hebt kun je er data visualisation in doen, maar het is behoorlijk traag en dit wordt zeker duidelijk op het moment dat je vele malen dezelfde acties moet gaan herhalen. Bovendien is het aardig lastig om een normale test in Unity te schrijven die gespecificeerd is voor agents.

1.1 Het is mij niet gelukt om een mooie agent based simulation op te zetten in Unity. Dit komt omdat ik persoonlijk de tutorial erg onduidelijk vond en nog nooit in Unity heb gewerkt. Ook heb ik nog nooit met C# gewerkt, maar dat was niet echt het struikelblok aangezien het heel erg lijkt op java.
   Zelf ben ik vrij lang bezig geweest om iets klaar te zetten voor presentatie, maar ik ben er gewoon nooit helemaal uit gekomen. Ik heb wel een character kunnen maken die kan rondbewegen in een 3D wereld, maar ik wou hier nog een object (in dit geval een cube) hebben die naar de player toe bewoog zodra de player binnen een bepaalde radius was. Dit is mij dus echter niet gelukt om te maken.

1.2 Unity heeft wel bestaande voorbeelden. Een voorbeeld hiervan zou een videogame character kunnen zijn. Dit character kan namelijk rondlopen in de wereld, maar hij zou bijvoorbeeld niet door een muur heen kunnen lopen. Er is code geschreven voor het character om bijvoorbeeld langs deze muur te lopen, zodat hij niet vast komt te zitten in een loop waarin hij tegen de muur aan blijft lopen. Hiervoor moet hij weten wat er om

hem heen is, aangezien hij namelijk de muren wel moet kunnen zien om hier mee te interacteren. Hij gaat van de staat ‘free roaming’ naar de staat ‘avoid wall’.

2. - Ii $0 \in I$
   Dit geeft eigenlijk aan welke mogelijke staten allemaal beschikbaar zijn in het programma. Dit kan handig zijn om er achter te komen wat een agent allemaal zou moeten kunnen. Het zou bijvoorbeeld kunnen dat een apparaat-agent een uit staat heeft, waardoor het alle benodigde functionaliteiten van een agent (rondkijken bijvoorbeeld) niet meer nodig heeft.

   - $S \rightarrow P$
   Dit geeft aan dat een agent een reactie kan geven op wat hij ‘ziet’. In het voorbeeld van mijn NPC die in een videogame rondloopt, kan hij dus een object zien dat zijn rechte-lijn pad blokkeert (zoals bijvoorbeeld een muur). Zodra hij dicht bij dit object is, ziet hij het object en past hij zijn gedrag aan. In dit geval zal de NPC besluiten om een tijdelijke nieuwe eindbestemming te pakken wat langs het object ligt, zodat hij op die manier langs het object kan lopen. Dit om te voorkomen dat hij recht in de muur blijft lopen (zoals in veel oude games gebeurde).

   - $I \rightarrow A$
   Dit staat voor een actie die volgt na een interne staatverandering. Dit kan bijvoorbeeld weer zijn dat een agent een object in zijn pad ziet. Bij de S, dan P hierboven was het enige wat de agent deed ‘perceiven’. Bij deze stap gaat hij ook actie ondernemen. Zo de agent dus een nieuw tijdelijk ‘eindpunt zetten’, en ook die kant op te gaan lopen om op die manier een object te ontwijken. Wat ook bijvoorbeeld kan is dat je in een videogame iemand zou kunnen beroven, en dat de gedupeerde achter je aan begint te rennen. Hij gaat over naar de interne staat ‘overvallen’, en past zijn gedrag aan als reactie.

   - $I \times P \rightarrow I$
   Dit geeft aan dat een agent van staat kan veranderen gebaseerd op in welke staat hij zich nu bevind. Dit kun je bijvoorbeeld het best zien in character ‘archetypes’ als een videogame die heeft. Zo kan bijvoorbeeld een character anders reageren op een actie die voor zijn ogen gebeurt. In het geval dat er bijvoorbeeld een character in zijn vision wordt belaagd, zal een superheld (bijvoorbeeld) anders reageren dan een ‘wimp’. De superheld zal overgaan naar de staat ‘probleem oplossen’ (waar het probleem het conflict in zijn vision is), terwijl een wimp misschien wel zou wegrennen.
   Dit kan ook toegepast worden op de eerdere staat van een agent. Als een agent bijvoorbeeld in een staat is waarbij hij van alles bang is, zal hij vluchten van een geval waarin hij anders bijvoorbeeld gewoon zou doorlopen.

3. Alle termen zijn beschreven met mijn project als voorbeeld. Dit was het bedoelde project, waarbij dus een kubus op de player af zou komen zodra de player binnen een bepaalde straal van de kubus was.

   Accessable – Inaccessable
   Het verschil tussen accessable en inaccessable is de hoeveelheid informatie die de agents hebben. Bij accessable heeft een agent ALLE informatie, bij inaccessable heeft de agent NIET

ALLE informatie. Als je vanuit de kubus in mijn opdracht kijkt, kun je dat berekenen door het weten van de afstand van de kubus naar de player. Als hij die niet ten alle tijden weet, is het al inaccessable. Mijn programma is accessable.

Deterministic – Non Deterministic
Het verschil tussen deze twee is de vraag of elke actie een gegarandeerde reactie heeft. Als er bijvoorbeeld kans in een agent zit, is je programma non deterministic. In mijn programma zal dat bijvoorbeeld zijn met het rondbewegen van de player. Als ik de vooruitloop-knop indruk, maar mijn agent zal niet vooruit bewegen (of heeft een kans om vooruit te bewegen), is het hele programma non deterministic. Een actie kan immers geen reactie geven. Dit heb ik echter niet gedaan. Mijn programma heeft alleen acties waarop een reactie voorkomen en is dus deterministic.

Episodic – Non episodic
Episodic vraagt zich eigenlijk af of het uitmaakt wat hij in een ander scenario zou doen. Dit kan je voor de uitleg niet echt goed teruglinken naar mijn programma, maar dit is wel bijvoorbeeld goed terug te linken naar als je een simulatie gaat doen over een kansspel. Bijvoorbeeld Bingo. Als hij in een simulatie bijvoorbeeld bal 42 als eerste trekt, dan zal hij niet tijdens de volgende simulatie met dezelfde code gaan zeggen dat bal 42 niet als eerst gepakt kan worden. In dat geval is het episodic.
Mijn programma zou non-episodic zijn.

Static - Dynamic
Static geeft aan dat de enige verandering in een environment komen van een agent. Dynamic geeft aan dat er meerdere dingen kunnen veranderen dan alleen de agent. Mijn programma zou dynamic zijn omdat de kubus van plaats kan veranderen gebaseerd op mijn agent. Als deze namelijk te dichtbij komt verplaatst de kubus.

Discrete - Continuous
Het verschil tussen deze 2 zit hem in de continuiteit van de simulation. Als een agent een bepaald, gedefineerd aantal stappen kan/moet zetten is hij discrete. In mijn programma zou je de agent oneindig lang rond kunnen laten lopen, wat betekent dat hij geen gedefineerde aantal stappen heeft en dus continuous is.

4. Een tegenovergesteld voorbeeld van mijn opdracht is **Monopoly**. Dit klassieke spel heeft een aantal grote verschillen van mijn simulatie. Het is namelijk inaccessable, deterministic en discrete. Terwijl mijn programma accessable, non-deterministic en continuous is.

Inaccessable:
Monopoly is inaccessable vanuit een agent gezien omdat een speler niet ten alle tijden weet wat er gebeurt op elk vakje. Er is namelijk een kans vakje waarbij de speler een kaart moet trekken. De speler weet voor het trekken (als het goed is) niet wat er op de kaart staat en kan deze informatie ook niet verkrijgen voordat hij de kaart trekt. Deze informatie is inaccessable en dus is het hele spel inaccessable.

Deterministic:
Monopoly is deterministic omdat ik net al zei dat er kans-kaarten in het spel zitten. Deze kaarten kunnen nutteloos zijn in de vorm van het spel omdat ze niks doen. Als een kaart de potentie heeft om niks te doen heb je een actie zonder reactie (kaart trekken – kaart doet verder niks). Dit betekent dat het spel deterministic is.

Discrete:
Monopoly is discrete omdat je de spelduur kan uitdrukken in stappen. De hoeveelheid stappen zijn van te voren niet te bepalen, maar na afloop kun je wel uitrekenen hoeveel stappen/beurten iedereen gezet heeft. Hierdoor is het niet een continuous spel maar een discrete spel.

Waarom is dit belangrijk?
Persoonlijk denk ik dat het wel belangrijk is om hier in ieder geval een keertje over na te denken. Als je namelijk het tegenovergestelde moet verzinnen van waar je nu mee bezig bent, moet je nadenken wat je agent allemaal is en kan, en wat het tegenovergestelde daarvan is en hoe dat eruit ziet. Zelf had ik eerst meerdere voorbeelden opgeschreven voordat ik werkelijk een voorbeeld kon geven wat aan 3 dichotomiën voldeed.