Door: Guy Veenhof

1.

Mesa, maakt gebruik van een Model waarin agents zitten. Het is agent-based, omdat de agent informatie kan opvragen van het model en dus dan een perceptie kan vormen. Je code understanding moet wel op een bepaald level zitten wil je goede code kunnen schrijven in python met mesa. Voor vrij simpele modellen moet je toch al aardig wat code schrijven om een model te visualizeren met JavaScript en Python. Het maakt gebruikt van browser based vizualisation. Het is wel van goeie kwaliteit en heel erg uitbreidbaar.

Voordelen:

1. Je hebt een mooie browser visualisatie
2. Alles is goeie kwaliteit en niet echt pixelated
3. Er is veel mogelijk met Mesa

Nadelen:

1. Veel code nodig voor matig visualisatie
2. Wel wat codeer ervaring nodig om echt wat goeds op te kunnen leveren. Je moet ook Javascript kunnen.

Elke agent heeft een wealth van 1 waar het mee begint. Vervolgens doe hij een stap in een van de 8 richtingen om zich heen.

Heeft de agent een wealth hoger dan 0 dan kiest het model een random ander agent die in een van de 8 vakjes om zich heen staat. De agent geeft dan 1 van z'n wealth aan de andere agent.

Heeft de agent een wealth van 0 dan kiest het weer een random ander agent die om zich heen staat en als die andere agent een wealth hoger heeft dan 3 dan neemt de agent 3 van de andere agent. Heeft de andere agent een wealth hoger dan 0 en kleiner dan 4 dan neemt de agent alleen 1 van de andere agent.

Ook word het gini coefficient bepaald en in een grafiek onder het grid gezet.

NetLogo, maakt gebruik van agents ook wel turtles genoemd. Het is een hele simpele en overzichtelijke manier van werken. Het is vrij ouderwets en niet echt uitbreidbaar. Omdat het heel simpel zijn veel projecten wel te maken, maar voor bedrijven is het niet echt aan te raden. Behalve als je een specifiek project hebt die compleet toepasbaar is op netlogo.

Unity, kan gebruik maken van agents. Het bied heel veel mogelijkheden waardoor je best goeie simulaties kan maken. Alleen Unity is niet echt gemaakt voor agent-based modelling. Het is meer gemaakt voor games. De leercurve is erg groot dus om snel iets te maken dat word moeilijk als je weinig tot geen ervaring hebt. Voor een agent-based simulatie is het eigenlijk te advanced omdat het in 3d word gedaan is het overkill.

2.

De agent kan eigenlijk in 2 soorten staten bevinden, het heeft geen wealth of het heeft wealth.

De "See/Percieve" function is wanneer de agent de omgeving waarneemt. Het model kijkt om zich heen of er andere agents zijn bij "cellmates" als hij van andere agents wealth afneemt of wealth geeft. Dit behoort tot "See/Percieve" function.

```python
def give_money(self):
    cellmates = self.model.grid.get_cell_list_contents([self.pos])
    if len(cellmates) > 1:
        other = self.random.choice(cellmates)
        other.wealth += 1
        self.wealth -= 1

def take_money(self):
    cellmates = self.model.grid.get_cell_list_contents([self.pos])
    if len(cellmates) > 1:
        other = self.random.choice(cellmates)
        # When an other
        if other.wealth > 3:
```

Zodra de agent een stap neemt word gekeken naar de 8 plekken om zich heen bij “possible_steps. Dit is een “See/Percieve” functie omdat het model voor de agent alleen naar de omgeving kijkt.

```python
def move(self):
    possible_steps = self.model.grid.get_neighborhood(
        self.pos,
        moore=True,
        include_center=False)
    new_pos = self.random.choice(possible_steps)
    self.model.grid.move_agent(self, new_pos)
```

Een “Act” function is wanneer een agent zich in een bepaalde staat bevind en een perceptie heeft van zijn omgeving kan de agent hierop een actie uitvoeren.

Wanneer een agent een stap heeft gedaan dan gaat het model kijken naar welke agents om zich heen bevinden en pakt een random agent. Als de agent zelf een wealth hoger heeft dan 0 geeft hij een wealth van zichzelf aan de andere agent. Dit is bij “give_money” functie. Heeft de agent een wealth van 0 dan neemt hij wealth af van een andere agent. Heeft de andere agent meer dan 3 wealth dan neemt de agent 3 wealth af van de andere agent. Heeft de andere agent tussen 4 en 0 wealth dan neemt de agent alleen 1 wealth af. Dit is te zien in de “take_money” functie Dit zijn “Act” functies

```python
def step(self):
    self.move()
    if self.wealth > 0:
        self.give_money()
    else:
        self.take_money()
```

```python
def give_money(self):
    cellmates = self.model.grid.get_cell_list_contents([self.pos])
    if len(cellmates) > 1:
        other = self.random.choice(cellmates)
        other.wealth += 1
        self.wealth -= 1

def take_money(self):
    cellmates = self.model.grid.get_cell_list_contents([self.pos])
    if len(cellmates) > 1:
        other = self.random.choice(cellmates)
        # When an other
        if other.wealth > 3:
            other.wealth -= 3
            self.wealth += 3
        # when
        elif 4 > other.wealth > 0:
            other.wealth -= 1
            self.wealth += 1
```

Een “Update” function is wanneer een agent een staat heeft en een perceptie en vervolgens verandert het in een nieuwe staat.

Het Model doet elke keer een stap en alle agents voeren hun schedule uit. Zodra de volgende stap word gedaan dan worden eigenlijk weer de buren gelezen van de agents en dan zijn de wealths van die agents verandert. Dit is eigenlijk de “Update” function.

```
class MoneyModel(Model):
    """A model with some number of agents"""
    def __init__(self, N, width, height):
        self.num_agents = N
        self.grid = MultiGrid(width, height, True)
        self.schedule = RandomActivation(self)
        self.running = True

        # Create agents
        for i in range(self.num_agents):
            a = MoneyAgent(i, self)
            # add a money agent to the schedule
            self.schedule.add(a)

            x = self.random.randrange(self.grid.width)
            y = self.random.randrange(self.grid.height)
            self.grid.place_agent(a, (x, y))

        self.datacollector = DataCollector(
            model_reporters = {"Gini": compute_ini},
            agent_reporters = {"Wealth":"wealth"}
        )

    def step(self):
        """goes step by step through the schedule"""
        self.datacollector.collect(self)
        self.schedule.step()
```

3.

De omgeving is vrij inaccessible, de agents kunnen namelijk niet alles uit de environment krijgen qua info. De agents kunnen alleen maar 1 vakje om hun heen kijken.

Het is non-deterministic, omdat er randomness in het model bevind die bijvoorbeeld de agent op een random manier naar een van de 8 richtingen te moven. Het bevind zich in een non-episodic omgeving omdat de acties de toekomstige acties wel kan veranderen. Als de agent zichzelf blut maakt dan heeft het geen wealth meer om te besteden, dus neemt hij wealth van anderen. Het is static, omdat als er geen volgende stap word uitgevoerd dan verandert er ook niets in het model zelf. Het is een continious, vanwege het feit dat er geen einde bestaat. Het kan blijven doorgaan met agents elkaar wealth laten geven en afpakken.

4.

De agents hebben 1 wealth. Zodra ze 1 wealth hebben. Dan doen ze niets meer. Dit is een nutteloze simulatie die deterministic is, omdat ze altijd klaar zullen zijn omdat ze 1 wealth hebben en niets

meer doen. Episodic, omdat het geen link heeft in wat de agent doet dat volgende scenarios kunnen veranderen. Discreet, omdat het meteen eindigt omdat elke agent met 1 wealth begint.