Intro to agents and agent-tools

Casper Smet

November 2019

1 Introduction

In this report I explain the Python library Mesa by illustrating an altered version of the tutorial simulation.

2 Robin Hood in the city

2.1 Original simulation

The simulation as given by the Mesa documentation is fairly simple. There are N agents interacting in a 2D space. This 2D space is called a MultiGrid. A (Multi)Grid is broken up into cells. In a MultiGrid each cell may contain zero or more agents.

Each agent starts with a *wealth* of 1. At each time step, an agent moves one space and, if it has any, gives 1 *wealth* to one of its neighbours.

2.2 Expanding on the original simulation

For this assignment one has to implement at least one new behaviour and element to the environment.

In the original simulation, *wealth* ended up being terribly distributed. The average Gini coefficient was upwards of 0.8. In the real world, the highest measurable Gini coefficient sits at about 0.7. This is a more than marginal difference in wealth distribution.

In order to fix this, money also needed to be taken from others. This is where the Robin Hood behaviour was implemented.

2.2.1 Steal from the rich, give to the poor(?)

If an agent has 0 *wealth*, they take 1 *wealth* from agents in the cells around them until they have as much *wealth* as there are agents in their cell.

The Robin Hood agent only takes *wealth* from agents who have more than 1 *wealth*

The Robin Hood agent then redistributes their newfound wealth in their home cell. Everyone in that cell receives 1 *wealth*, including the Robin Hood agent.

If the agent cannot find as many units of *wealth* it needs, it returns to its own cell. It then gives away the units of *wealth* it had managed to take from other agents.

This new type of behaviour took the Gini coefficient down to about 0.4. A significantly fairer distribution of wealth.

2.2.2 Introduction of cities

People eventually stop moving around and congregate in small areas. This is why the city element was implemented into the environment. The introduction of this element also necessitates the introduction of some new behaviour.

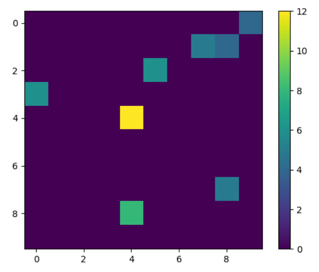
A city is defined as a cell with more than 2 agents inhabiting it. Due to the superior defences of a city, a Robin Hood agent is unable to steal from a city.

This does not mean, however, that an agent in a city is safe from any type of *wealth* loss. If agent A in a city has 0 *wealth* and encounters agent B in the same cell with more than 0 *wealth*, agent A will take 1 *wealth* from agent B.

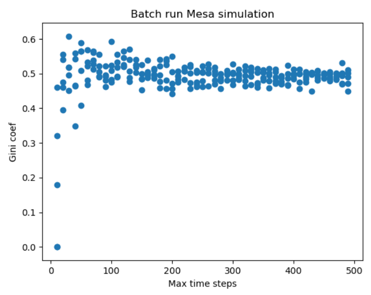
Finally, when an agent enters a city cell, they stop moving. Instead, the agent stays in the city indefinitely.

The added element of cities, and the behaviour that comes along with it, increased the Gini coefficient. Wealth distribution is, on average, significantly worse than before. It now sits at an average of 0.5.

This can be attributed to the fact that Robin Hood behaviour slowly dies out. Eventually, only cities remain.



(a) An image of a MultiGrid showing agent densities (cities)



(b) Gini coefficients after batch run

2.2.3 Agent-oriented programming

This simulation was written with *mentalistic* notions in mind. The idea that each agent has a belief, desire and intention.

Each agent wishes to have *wealth*. They also wish for their communities to have *wealth*. This is why they freely give *wealth* to other agents. Through sheer chance, some agents end up having more *wealth* than other.

An agent's first priority is itself. If they do not have even the barest amount of *wealth*, they will take it from another.

An agent also wishes to protect its *wealth* from others, which is why it congregates with others. Strength in numbers.

These can be condensed into three ideas:

Belief	All agents ought to have *wealth*
Desire	I wish to have at least 1 *wealth*
Intention	I will redistribute *wealth*, as long as it does not hurt me

While the paradigm was not followed into writing the actual code, it was kept in mind for designing the simulation. The code itself is more reminiscent of Object-Oriented Programming.

3 Recommending Mesa

Mesa is a very powerful module. This is can be attributed to a couple of things:

3.1 Cellular simulations

Mesa is especially powerful for cellular simulations. Its space module is focused on a cellular representation of space (Grids) and has all the features one would need to properly utilise them.

While a lot of simulations can be built in a cellular style, not all types simulations are suitable for this.

For this problem, Mesa offers the ContinuousSpace class. As one would guess, this can be used to simulate continuous space. While this is a perfectly use able class, it does not offer as many luxuries as Mesa's grid class.

3.1.1 Unity

Unity delivers similar luxuries as Mesa does for continuous space, but falls flat in comparison to Mesa's cellular space. It does, however, have the benefit of being able to simulate 3D environments. Something which Mesa, to my knowledge, cannot.

3.1.2 NetLogo

NetLogo, as a tool, was built from ground up to be an agent-based simulation tool. It has most of Mesa's luxuries for cellular simulations, but no support for continuous space.

3.2 Python

The fact that Mesa is a Python module is in-and-of-itself a benefit. For one, Python as a language is easy to read and write. Secondly there are countless libraries in Python. Any difficult calculations or analyses can be made trivial simply by using the correct library.

3.2.1 Unity

All programming in Unity is done in C#. While this language is also an industry standard, it does not have the inherent versatility of Python. Everything can be done in C#, but "There's a library for that" cannot be said quite as often as in Python.

Any data collection or analysis of the simulation would most likely also have to be done in a separate format from the simulation itself, whereas in Python it can be done simultaneously with the simulation itself.

3.2.2 NetLogo

NetLogo uses its own proprietary scripting language. While fairly simple, it would still take time to learn. It is also far from an industry standard, as it cannot be used outside of NetLogo.

Data collection and analysis tools are built into the language and application.

3.3 Visualisation

Due to the open nature of Mesa, one can use any Python visualising tool one would like to. A visualisation could be fully built into the simulation, or could be generated afterwards. Mesa also offers a web framework library, making implementing interactive visualisations child's play.

3.3.1 Unity

Unity is an inherently visual tool. One can fully explore the environment's space. It is, however, not good at visualising the data collected from a simulation. This would most likely need to be done externally.

3.3.2 NetLogo

NetLogo is intended only as an agent-based simulation tool. Unlike Mesa, it is inherently a visual tool. It also includes built in data visualisation tools. While NetLogo's visualisation tools aren't as customisable as Python's, they are easy to use and uniform.

3.4 Conclusion

I would recommend using Mesa as ABMS tool. It is highly versatile, and most importantly, it is a Python library. The ability to use Python adds an incredible amount of value. NetLogo is clearly the second choice. NetLogo has all of the tools necessary to create and analyse simulations. The requirement to use its proprietary language and its lack of customisability are both its greatest and weakest points.

While Unity might be better for simulating physics and can do 3D, the lack of data collection and analysis tools make it too unwieldy of a tool. Unity was never meant to be a simulation tool, it is a physics engine for video games.

4 Describing through states and functions

In this section and the next section I extrapolate on section 2 **Robin Hood in the city**. In this section, I describe the simulation through the behaviour of the agent, using terms such as perceive, act and update.

4.1 Generic loop

Each of the agent's 'steps' can be defined as such:

They perceive (P) a certain amount of information based on the state of their environment (S). This is known as see or perceive.
$S \to P$

Based on what they perceive, their internal state changes. This is known as update.
$I \times P \to I$

Based on their current internal state (I) they take an action (A). This is known as act.
$I \to A$

4.2 Defining the loop

An agent knows only two things about itself. It learns these things throught the update function.

- If it has more than 0 *wealth*
- If it is currently in a city[1]

Initially, an agent's state can be defined as a power set[2] of these two propositions. I then has a cardinality of 4. After the update functions, it learns the following things:

- How many cellmates it has
- For all of its cellmates, if they have more than 0 *wealth*
- For all of its neighbours in a two cell radius, if they have more than 1 *wealth*
- For all of its neighbours in a two cell radius, if they are currently in a city

[1]This bit of state would better fit environmental state as it is based on how many cellmates the agent has. It could be learned through the update function.

[2]Note including empty list or zero elements

This increases the cardinality of I significantly. Each cell could hold N agents. If my calculation is right, that means the cardinality is $(10N + 4)^2$. In this simulation N=50, that means a cardinality of 254016.

The majority of I, however, does not matter for the act functions. The environment states related to an agent's neighbours are only taken into account when the agent itself is *not* in a city. This greatly simplifies things.

I can be simplified even further, though that does require moving away from set-theory. Imagine the following set definition, with some pseudo code list comprehensions: $(m_w, m_c, c_c, [..c_{wn}], [..o_{wn}], [..o_{cn}])$

Where:

$$m_w \leftarrow \textbf{If agent has more than 0 units of } \textit{wealth} \tag{1}$$

$$m_c \leftarrow \textbf{If agent is currently in a city} \tag{2}$$

$$c_c \leftarrow \textbf{How many cellmates the agent currently has} \tag{3}$$

$$[..c_{wn}] \leftarrow \textbf{For all cellmates if they have more than 0 units of } \textit{wealth} \tag{4}$$

$$[..o_{wn}] \leftarrow \textbf{For all neighbours if they have more than 1 unit of } \textit{wealth} \tag{5}$$

$$[..o_{cn}] \leftarrow \textbf{For all neighbours if they are in a city} \tag{6}$$

So for $I \rightarrow A$, where I equals $(0, 1, 1, c_{w0} = 1, ..., ...)$, the action A equals taking one wealth from c_{w0} and adding it to its own. The next time the update function is run, I will be updated to $(1, 1, 1, c_{w0} = 0, ..., ...)$

5 Dichotomic description

In this section I describe the created environment based on the dichotomies as described by Woolridge & Jennings in Intelligent Agents p.6.

5.1 Accessible vs. inaccessible

Accessibility is defined by how complete, accurate and up-to-date the information about an environment's state can be obtained the agent. This is an almost entirely accessible environment. The agent perceives if its current cell is a city, it can see how much *wealth* [3] another agent has in comparison to a set amount[4] and it can perceive how many agents share its current cell.

There are two limiting factors:

- It cannot see the exact amount of *wealth* of any agent
- It can only compare wealth with agents in its cell, or in a radius of two cells when using Robin Hood behaviour

[3]Technically this is an agent's state, not the environment.

[4]More than 0 or more than 1

5.2 Deterministic vs. non-deterministic

A deterministic environment ensures that any action has a single guaranteed effect. This is mainly the case in this simulation. An action A_1 at state s_1 always results in state s_2. When an agent has 0 *wealth*, lives in a city and its cellmate has 1 *wealth* (s_1), the agent takes wealth from its cellmate (A_1), resulting in the agent's *wealth* increasing to 1 and its cellmate's decreasing to 0 (s_2).

There is one non-deterministic aspect to this simulation: which cellmate an agent gives 1 unit of its *wealth* to. Or which cellmate's unit of *wealth* gets stolen. Both are decided not by any state, but by a random number generator.

5.3 Episodic vs. non-episodic

In an episodic environment an agent does not need to reason about the interactions between this and future episodes. It simply acts on the current state of things. This environment is episodic, the agent acts purely on the current state, it does not take previous or future states into account.

5.4 Static vs. dynamic

A static environment is one that is only influenced by the actions of the agent, whereas a dynamic environment also has other influences acting on it.

This environment is somewhat difficult to classify. Outside of the agents, no other processes operate on it[5]. An individual agent may, however, influence the state of another agent. The actions of the prior agent are beyond the control of the latter agent.

5.5 Discrete vs. continuous

A discrete environment is an environment with a fixed, finite number of actions and percepts in it. This environment leans more towards discrete rather than continuous.

The state of each agent is defined by two things, *wealth* and if it is in a city. The prior having a set max[6] and minimum[7], the latter being a bool. A finite number of states ensures a finite number of percepts.

An agent has but 3 abstract actions it can take depending on its state and the state of the world around it. This, in turn, ensures that there are a finite number of actions.

[5]One could argue that the cities being defined in the 'model' step function instead of the agent 'step' function indicates an outside process. It was not intended to be

[6]Number of agents

[7]0

5.6 Opposites

5.6.1 Accessible vs. inaccessible

While it is certainly useful for the simulation that the agent knows as much as it does about its surroundings, it would make for sense for Robin Hood behaviour to know less.

Robin Hood behaviour only works on neighbours that are not in cities. These agents are per definition travellers. It would be highly unlikely that the Robin Hood agent actually knows where these travellers are. The agent should not know.

This abstraction of knowledge makes the environment considerably less accessible.

5.6.2 Episodic vs non-episodic

Realistically an agent would think about which agent it steals *wealth* from. If agent A had previously been given *wealth* by agent B, it would not make sense for A to then steal more *wealth* from B.

This functionality would require being able to look back on previous state, making the environment less episodic.

5.6.3 Deterministic vs. non-deterministic

Finally, it is not realistic for a theft to always succeed. In a more realistic model, agents would defend themselves from theft. This does, however, require some influence of chance.

The introduction of chance, an influence outside of state, would make the environment non-deterministic.