Simulations on social networks

Agent-based modelling, Konstanz, 2024

Henri Kauhanen

25 June 2024

Plan

- Last time we learned about (social) networks
- Today, we'll learn how to interface Graphs.jl with Agents.jl
- This allows ABM simulations on networks: our population of agents can now be structured in very interesting ways
- We'll also talk about how to gather statistics from simulation runs
- For a concrete example, we will implement a simulation with variational learners on a social network without spatial effects

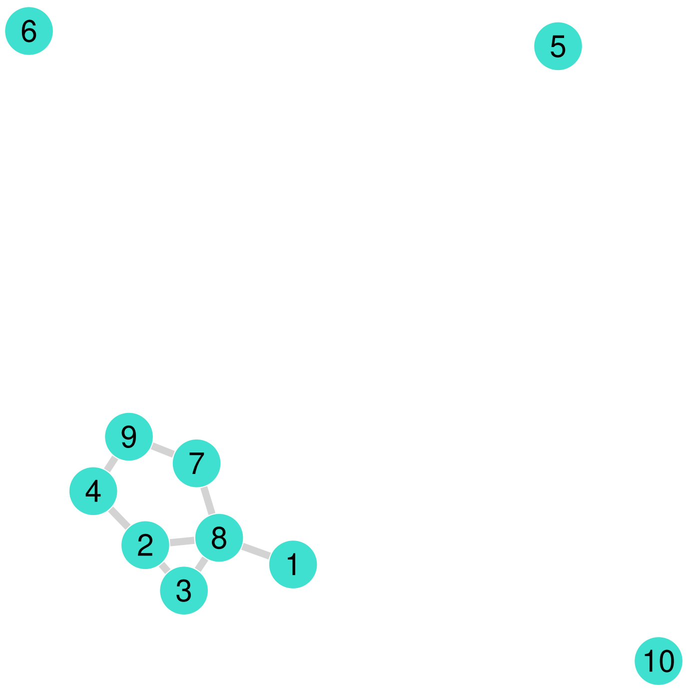
Requirements

- We will need the following packages today:
  - Agents
  - Graphs
  - Plots
  - Statistics
  - DataFrames
- Now would be a good time to make sure you have all these installed
- Code for today's lecture: VL2.jl under "Bric-a-brac"

Catching errors

- Before moving on, we need to discuss an important technicality
- Sometimes, your code throws an error
  - For example, try: `rand([])`

  - This tries to draw a random element from an empty array, which of course won't work
- Often, you want to eliminate these errors
  - For example, make sure that you never use things like `rand([])`
- Other times, they may be unavoidable, and need to be **caught**



- For a concrete example, assume the above network
- Suppose we want to obtain a random neighbour of a given node
- If that node does have neighbours, then all is well
- But if the node happens to have no neighbours (such as node number 5), then we have a problem!
- If we don't want our code to crash, we need to catch the error

- Errors are caught using a `try ... catch ... end` block:

```
try
  rand([])
catch
  println("Trying to draw from an empty container!")
end
```

```
Trying to draw from an empty container!
```

- Here, Julia will try to execute everything found between the `try` and `catch` keywords
- If an error is thrown, then it is caught and the stuff between the `catch` and `end` keywords is executed, **without crashing**

- You can also leave the catch block empty, if you just want to continue silently!

```
try
  rand([])
catch
end
```

Strategy

- Recall the 5 steps to define a model in Agents.jl:

  1. Decide on model space
  2. Define agent type(s)
  3. Define rules that evolve the model
  4. Initialize your model
  5. Evolve, visualize and collect data

- For this particular application:

  1. Decide on model space — we'll use a graph
  2. Define agent type(s) — reuse existing code, with small modifications
  3. Define rules that evolve the model — reuse, with small modifications
  4. Initialize your model — just like before
  5. Evolve, visualize and collect data — we'll talk more about this

1. Space

- In a previous lecture, we used:

```
dims = (10, 10)
space = GridSpace(dims)
```

- Now, we use (for example):

```
G = erdos_renyi(100, 0.3)
space = GraphSpace(G)
```

2. Agent

- Previously, we defined:

```
@agent struct GridVL(GridAgent{2}) <: VariationalLearner
  p::Float64
  gamma::Float64
  P1::Float64
  P2::Float64
end
```

- We now add a new type:

```
@agent struct NetworkVL(GraphAgent) <: VariationalLearner
  p::Float64
  gamma::Float64
  P1::Float64
  P2::Float64
end
```

3. Rules

- We previously used:

```
function VL_step!(agent, model)
  interlocutor = random_nearby_agent(agent, model)
  interact!(interlocutor, agent)
end
```

- Here, `random_nearby_agent` returned the 8 agents surrounding `agent` in the `GridSpace`

- `random_nearby_agent` also has a method for `GraphSpaces`
- However, in a graph, an agent may be neighbourless!
- We need to consider this, and so define:

```
function VL_step!(agent::NetworkVL, model)
  try
    interlocutor = random_nearby_agent(agent, model)
    interact!(interlocutor, agent)
  catch
  end
end
```

4. Initialize model

- Previously, we used the following to initialize a model:

```
model = StandardABM(GridVL,
                    GridSpace((10, 10)),
                    agent_step! = VL_step!)
```

- Now we do (for example):

```
model = StandardABM(NetworkVL,
                    GraphSpace(erdos_renyi(10, 0.5)),
                    agent_step! = VL_step!)
```

Putting it all together

```
# create space
G = erdos_renyi(10, 0.5)
space = GraphSpace(G)

# initialize model
model = StandardABM(NetworkVL,
                    space,
                    agent_step! = VL_step!)

# add agents
for i in 1:10
  add_agent_single!(model, 0.01, 0.01, 0.4, 0.1)
end
```

5. Evolve, visualize and collect data

- Previously, we used the `step!` function from Agents.jl to evolve our model
- We also wrote our own custom functions for retrieving summary statistics
- Graphs.jl actually contains **data collection** functions which make this even easier
- The most important of these (for us) are `run!` and `ensemblerun!`

Using `run!` to collect data

- `run!` is like `step!`, except it also collects the model's state and returns it as a `DataFrame`
  - `DataFrame`s are tables that are used to represent data
- Syntax: `run!(model, n; adata)`, where
  - `n` is the number of time steps we want to run the model for
  - `adata` is a keyword argument that specifies what data ("agent data") is gathered
- For reasons which are not superbly clear, `run!` has *two* return values, two `DataFrame`s
  - We can safely ignore the second one
- Example: suppose we want to collect each agent's `p` field (their weight for grammar G_1) at each time step, over 4000 model iterations (each agent is updated 4000 times).
- This is achieved by:

```
data, _ = run!(model, 4000; adata = [:p])
```

- Note the two return values; since we are not interested in the second one, we store it in the _ variable (think of this as a trash bin)
- Also note that `p` is prefixed with a colon (`:p`) – this is crucial!
- Also note that `adata` is a vector (this means you can specify more than one agent field to be collected, should you wish to do so)

> **Note**
>
> Why does `p` have to be prefixed by a colon? Well, we couldn't just put `p` in there, as that would refer to a variable whose name is `p`. But that's not what we want. What we want is somehow to refer to each agent's internal `p` field. This can be achieved using so-called **symbols**. In Julia, symbols always begin with a colon (`:`). Think of them as labels. They are a bit like strings, but not quite (for example, a string is composed of characters but a symbol isn't!).

- The variable `data` now contains a `DataFrame` with three columns: the time step, the agent's ID, and the agent's `p`:

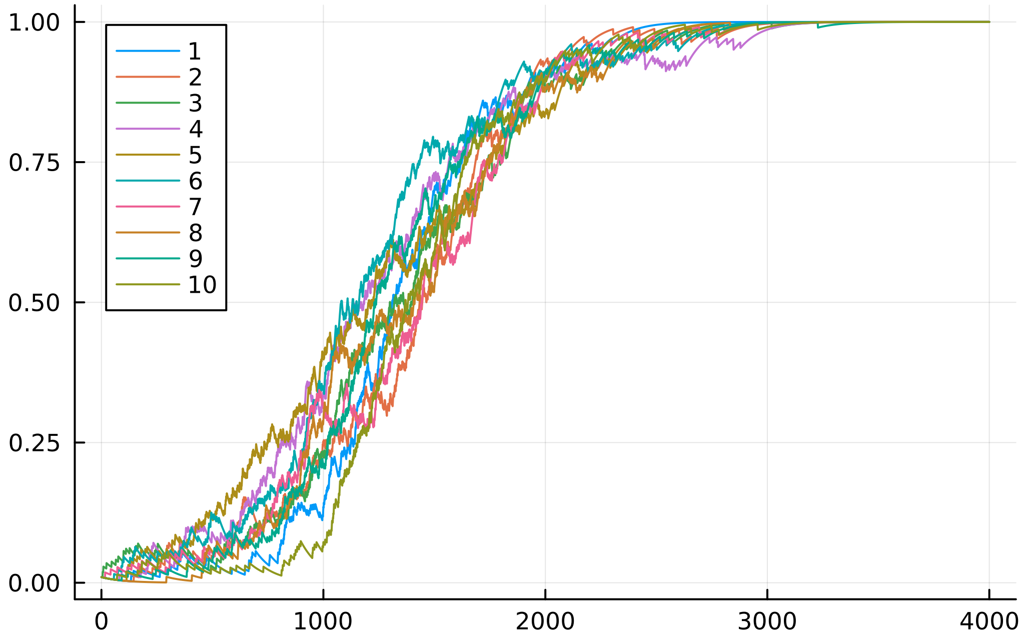
```
data
```

	time	id	p
	Int64	Int64	Float64
1	0	1	0.01
2	0	2	0.01
3	0	3	0.01
4	0	4	0.01
5	0	5	0.01
6	0	6	0.01
7	0	7	0.01
8	0	8	0.01
9	0	9	0.01
10	0	10	0.01
11	1	1	0.0099
12	1	2	0.0099
13	1	3	0.0099
14	1	4	0.0099
15	1	5	0.0099
16	1	6	0.0099
17	1	7	0.0099
18	1	8	0.0099
19	1	9	0.0099
20	1	10	0.0099
21	2	1	0.009801
22	2	2	0.009801
23	2	3	0.009801
24	2	4	0.009801
25	2	5	0.009801
26	2	6	0.009801
27	2	7	0.009801
28	2	8	0.009801
29	2	9	0.009801
30	2	10	0.009801
...	...	...	...

- We can now, for example, plot this:

```
using Plots
plot(data.time, data.p, group=data.id)
```

- Here, note that:
  - columns of a data frame are selected using the dot (`.`)
  - we use the `group` keyword argument on `plot` so that each agent gets its own trajectory in the plot



Collecting aggregated data

- Often we don't need to collect data for each agent individually
- For example: it is often enough to know how the average, or mean, of `p` evolves
- This is very easy to do with `run!`: all we need to do is to feed the `adata` keyword argument with the tuple `(:p, mean)` instead of plain `:p`
- More generally, in place of `mean`, you can put any function that you want to aggregate over the agents

- Example:

```
using Statistics

G2 = erdos_renyi(10, 0.5)
space2 = GraphSpace(G2)

model2 = StandardABM(NetworkVL, space2,
                     agent_step! = VL_step!)

for i in 1:10
  add_agent_single!(model2, 0.01, 0.01, 0.4, 0.1)
end
```

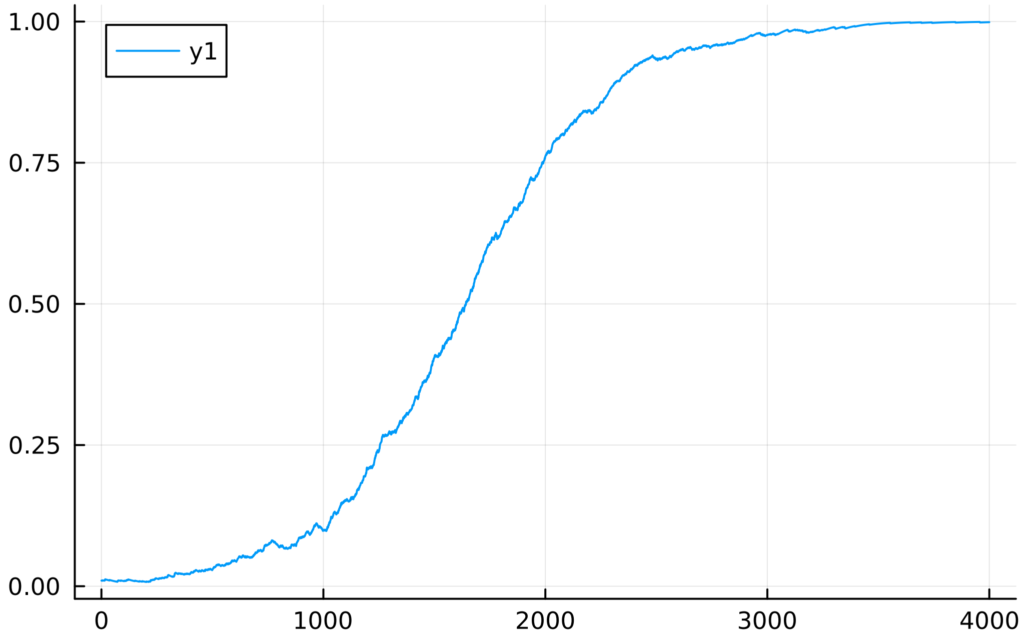
```
data2, _ = run!(model2, 4000; adata = [(:p, mean)])
```

- Now the returned dataframe contains a `mean_p` column:

```
data2
```

	time	mean_p
	Int64	Float64
1	0	0.01
2	1	0.0099
3	2	0.009801
4	3	0.00970299
5	4	0.00960596
6	5	0.0105099
7	6	0.0104048
8	7	0.0103008
9	8	0.0101977
10	9	0.0100958
11	10	0.00999481
12	11	0.00989486
13	12	0.00979591
14	13	0.00969795
15	14	0.010601
16	15	0.010495
17	16	0.01039
18	17	0.0122861
19	18	0.0121633
20	19	0.0120416
21	20	0.0119212
22	21	0.011802
23	22	0.011684
24	23	0.0115671
25	24	0.0114515
26	25	0.0113369
27	26	0.0112236
28	27	0.0111113
29	28	0.0110002
30	29	0.0108902
...	...	...

```
plot(data2.time, data2.mean_p)
```



Exercise

- Recall that the constructor of a Watts–Strogatz network, `watts_strogatz(n, k, beta)`, takes three arguments:
  - `n`: number of nodes
  - `k`: initial degree
  - `beta`: rewiring probability
- Your task: simulate VLs in a Watts–Strogatz network, exploring how/if variation in `beta` affects the evolution of mean `p`
- Use these parameters for your learners:
  - initial value of `p`: 0.01
  - learning rate `gamma`: 0.01
  - `P1`: 0.2
  - `P2`: 0.1
- Use these for your network(s):
  - `n`: 50

- k: 8
- beta: 0.1 versus 0.5

Solution

It is useful to wrap the model construction in a function:

```
function make_model(beta)
  G = watts_strogatz(50, 8, beta)
  space = GraphSpace(G)

  model = StandardABM(NetworkVL, space,
                      agent_step! = VL_step!)

  for i in 1:50
    add_agent_single!(model, 0.01, 0.01, 0.2, 0.1)
  end

  return model
end
```

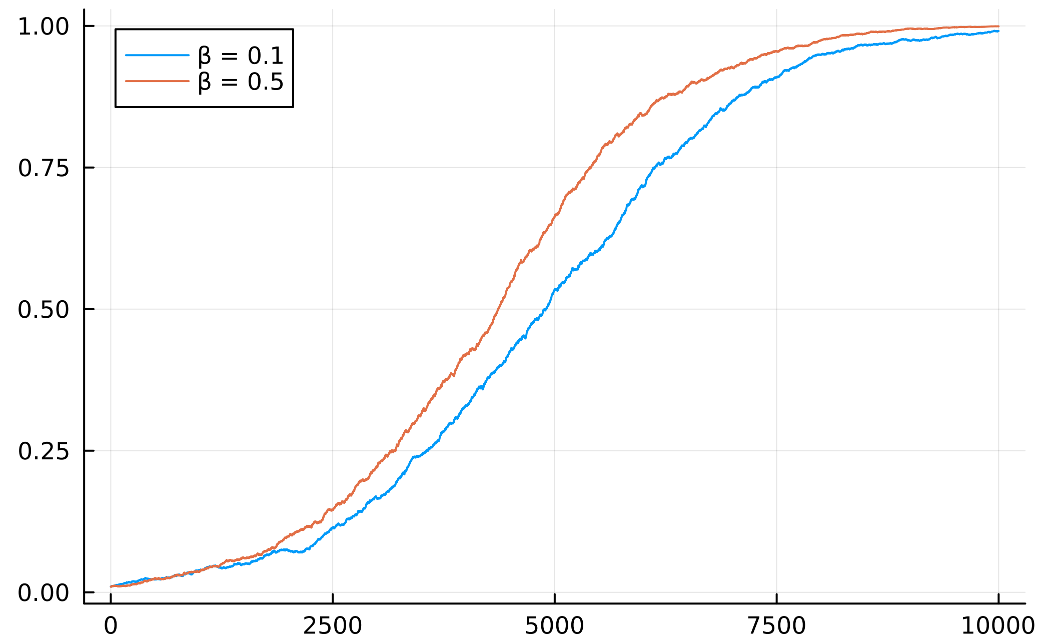
It is now easy to run both models:

```
model1 = make_model(0.1)
model2 = make_model(0.5)

data1, _ = run!(model1, 10_000; adata = [(:p, mean)])
data2, _ = run!(model2, 10_000; adata = [(:p, mean)])

plot(data1.time, data1.mean_p, label="  = 0.1")
plot!(data2.time, data2.mean_p, label="  = 0.5")
```

And to visualize the results:



Ensemble data with `ensemblerun!`

- It looks like there might be a small difference: the change is quicker with $\beta = 0.5$ compared to $\beta = 0.1$
- But is this difference real, or just a random fluke?
- To answer this question, we need to run several repetitions of each simulation!
- This is easiest by using the dedicated `ensemblerun!` function
- It works like `run!` but, instead of a single model, takes a *vector* of models as input

- Like this:

```
models1 = [make_model(0.1) for i in 1:10]
models2 = [make_model(0.5) for i in 1:10]

data1, _ = ensemblerun!(models1, 10_000; adata = [(:p, mean)])
data2, _ = ensemblerun!(models2, 10_000; adata = [(:p, mean)])
```

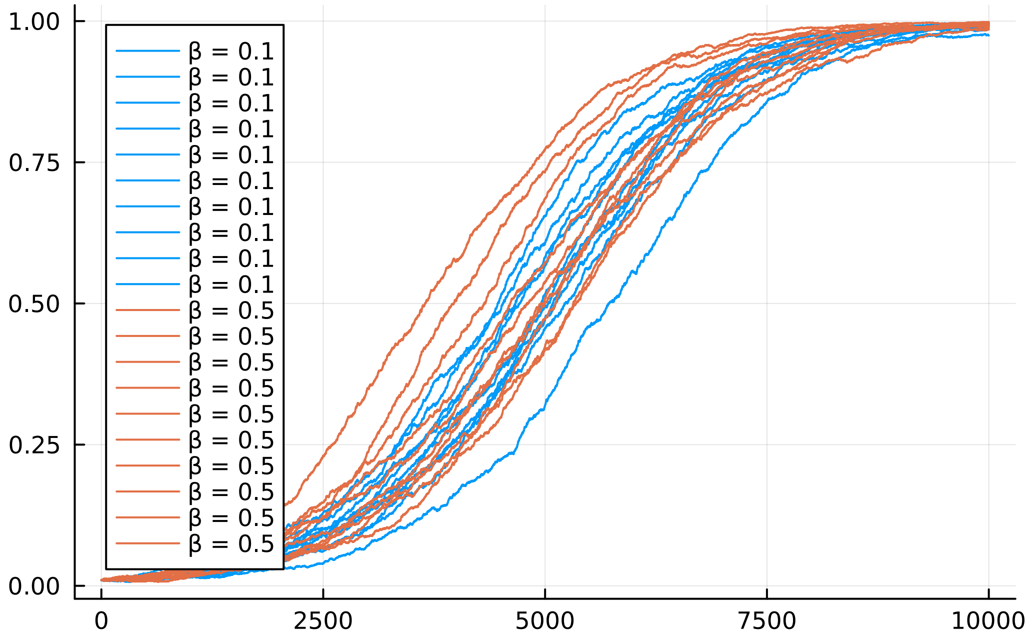
- The returned data frames look like this:

```
data1
```

	time	mean_p	ensemble
	Int64	Float64	Int64
1	0	0.01	1
2	1	0.0101	1
3	2	0.010199	1
4	3	0.010297	1
5	4	0.010194	1
6	5	0.0100921	1
7	6	0.00999118	1
8	7	0.00989127	1
9	8	0.00999235	1
10	9	0.00989243	1
11	10	0.00979351	1
12	11	0.00969557	1
13	12	0.00979862	1
14	13	0.00970063	1
15	14	0.00960362	1
16	15	0.00950759	1
17	16	0.00961251	1
18	17	0.00951639	1
19	18	0.00942122	1
20	19	0.00952701	1
21	20	0.00943174	1
22	21	0.00933742	1
23	22	0.00924405	1
24	23	0.00915161	1
25	24	0.00906009	1
26	25	0.00896949	1
27	26	0.0088798	1
28	27	0.008791	1
29	28	0.00870309	1
30	29	0.00881606	1
...	...	...	...

- To plot these in a sensible way, we need to group by the ensemble column:

```
plot(data1.time, data1.mean_p,
     group=data1.ensemble, color=1, label="  = 0.1")
plot!(data2.time, data2.mean_p,
      group=data2.ensemble, color=2, label="  = 0.5")
```



- These results suggest that there may be a difference; however, it looks to be small
- To settle this question more conclusively, we need to:
  - Run more simulations!
  - Do some statistics on the simulation results
- You get to practice both these things in this week's homework