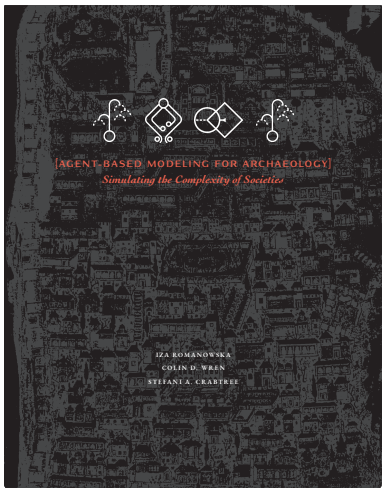
PLEASE NOTE:

The contents of this open-access PDF are excerpted from the following textbook, which is licensed under a Creative Commons Attribution-ShareAlike 4.0 International License:

Romanowska, I., C. D. Wren, and S. A. Crabtree. 2021. *Agent-Based Modeling for Archaeology: Simulating the Complexity of Societies.* Santa Fe, NM: SFI Press.

This and other components, as well as a complete electronic copy of the book, can be freely downloaded at https://santafeinstitute.github.io/ABMA

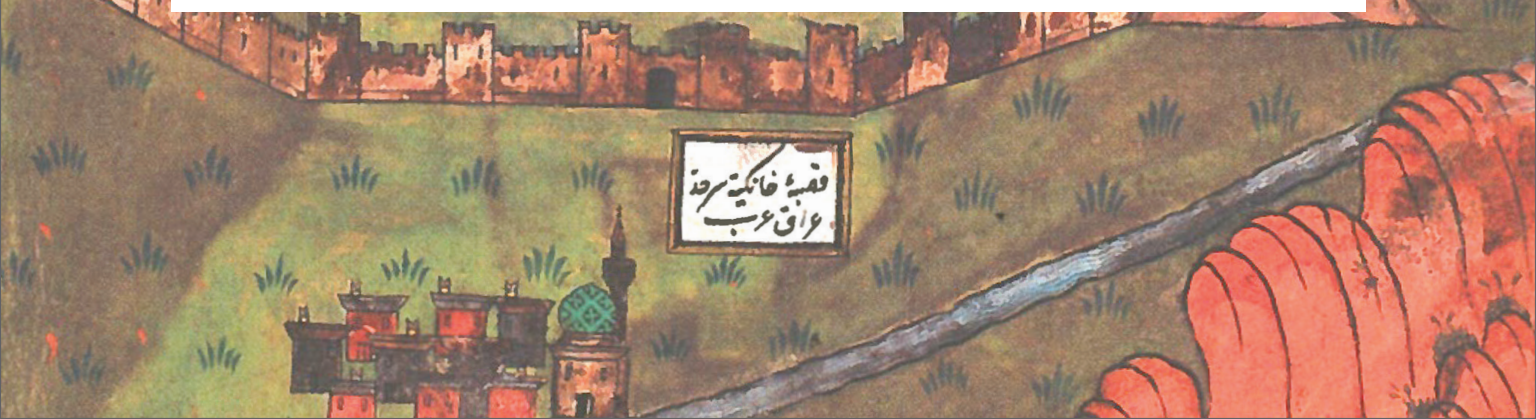


REGARDING COLOR:

The color figures in this open-access version of *Agent-Based Modeling for Archaeology* have been adapted to improve the accessibility of the book for readers with different types of color-blindness. This often results in more complex color-related aspects of the code than are outlined within the code blocks of the chapters. As such, the colors that appear on your screen will differ from those of our included figures. See the "Making Colorblind-Friendly ABMs" section of the Appendix to learn more about improving model accessibility.

SFI PRESS

THE SANTA FE INSTITUTE PRESS 1399 Hyde Park Road, Santa Fe, New Mexico 87501 | sfipress@santafe.edu



MODELING WITH RELATIONAL DATA: RELATIONSHIPS AND EXCHANGE USING NETWORK SCIENCE

8.0 Introduction

OVERVIEW
- ▷ Theory to data modeling spectrum
- ▷ Working with relational data
- ▷ Tutorial on NetLogo's Network extension
- ▷ Fundamentals of network science
- ▷ Testing code

In chapter 1, we mentioned the three main functions of modeling: hypothesis testing, data exploration, and theory building. Often there are no firm boundaries between these, and most models combine these different functions. In some cases archaeologists may want to understand the past of a specific small community—for example, traders in a provincial Roman town. Other times, we may want to uncover regularities defining human groups that span time and space—for example, the impact of taxation on large-scale integrated economic systems. Both of these examples will use theory and data; both will test hypotheses. However, the Roman case will be much more precise (e.g., the full size of the town can be modeled) and perhaps realistic (e.g., traders will have limited information). In the second case the model may be more abstract, making the results more generalized, that is, applicable to many different countries and time periods and potentially more tractable, meaning that the dynamics and interactions are better understood.

tractability: a feature of models denoting how easy it is to understand the internal dynamics, interactions between processes, and causality chain that leads to the patterns in output.

Bullock (2014) has visualized the trade-offs among realism, generality, precision, and **tractability** that all models face (fig. 8.0). Any one model cannot be highly realistic and precise while also covering a wide range of cases. Think of a map: very precise hiking maps are also highly figurative and never cover a large area, while the realistic (e.g., satellite) map of your entire country would leave you scrambling to find the nearest supermarket or even some of the smaller towns. Similarly, it is not possible to create a simulation that is realistic, gives precise predictions, and covers a wide range of cases. The final direction on the trade-off triangle—tractability—denotes our ability for to understand the mechanisms and explain the results. This means not just showing that factor

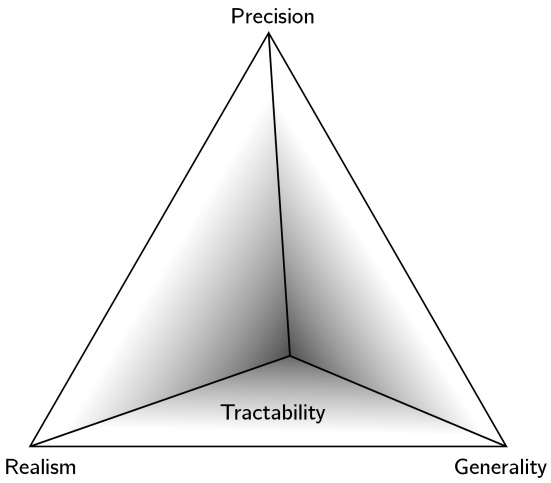


Figure 8.0. The modeling trade-off between precision, realism, generality, and the additional dimension of tractability. It is not possible to optimize all factors, so the modeler must decide which dimension to sacrifice. Adapted from Bullock (2014, fig. 3).

X increases output Y but being able to trace, step by step, the chain of causality that leads from one to the other, therefore bringing to light the full mechanism driving this relationship. Highly realistic models are often not highly tractable because they usually serve a different purpose, such as prediction. On the other hand, simple models usually have easily understandable dynamics but may not be that useful in the real world (Taghikhah, Filatova, and Voinov 2021).

It's easy to add new parameters and processes into a model to increase its realism, but reckoning comes when the modeler needs to explain the results. Nontractable models are often referred to as "black-box" models.

data-driven models: models maximizing realism and precision, incorporating detailed data and often used to predict future outcomes.

Data-driven models geared toward prediction are often designed with one case in mind—they often maximize precision and realism, yet are difficult to generalize toward other systems and may be low in tractability, especially when the mechanisms driving the modeled processes are not well known. For example, in public health, agent-based models built on theories and data of human behavior are used to predict outcomes of the spread of specific diseases, as we recently have seen in the COVID-19 pandemic (Lorig, Johansson, and Davidsson 2021). They often include realistic GIS maps, detailed population data, and very specific characteristics of a certain disease. These highly precise and realistic models may not be easy for the lay reader to unpack due to the elaborate combination of algorithms and parameters, yet even they are often built up from the simple theoretical framework of SIR models. In-

Recall the emulation and explanation types of models defined by Premo we discussed in 6.

depth research, model building, and testing are required to create these highly intricate (precise and realistic) agent-based models (Hammond et al. 2021).

On the other end of the spectrum, physicists are famous for constructing models maximizing generality and paying less attention to the applicability of these abstract representations to "the real world." A famous joke says that, given a chance, a physicist will start their inquiry with the statement "consider a perfectly spherical cow in a vacuum . . ." This is not to say that other disciplines do not engage with highly general theory-building models; when developing a hypothesis, all scientists need a tool to understand the repercussions of the proposed mechanism. **Theory-driven** models (sometimes called "**subjunctive**" or "what-if" models) provide predictions for a range of outcomes that can help refine our understanding and direct future research.

Check out *Spherical cow* on Wikipedia. It is an established scientific concept.

theory-driven models: models maximizing generalism, usually exploring relationships and causality and often used to explain phenomena.

Subjunctive models tackle the question, "What would happen if . . . ?"

Many archaeological simulations fall in the category of explanatory models, where the underlying dynamics of the system are not certain or can even be entirely unknown, yet we want to explain the endpoint of a process. Tractability of these kinds of models is therefore important since we want to answer the question "What is the explanation for . . . ?" This means that understanding the ways that parameters interact is paramount and finding the most parsimonious model is key for developing understanding. The second most common category of archaeological simulations are models that test hypotheses. These tend toward realism and precision. They can be focused on testing what mechanism was at play or examining how particular circumstances led to specific events in the past. In both cases, tractability is key. Here, realism is necessary to establish plausible causal links, while a certain level of precision is indispensable so that the results can be compared with archaeological data.

Explanation lies at the heart of the majority of archaeological models. Our research questions usually begin with "Why?" or "How?"

Although there is no prescribed way to evaluate whether a model has the right level of complexity built into it, it is worth keeping in mind at all times the well-known statement attributed to Albert Einstein (Sessions 1950): "Everything should be as simple as it can be, but not simpler." We discussed parsimony in more depth in chapter 5, since the concept is important for developing a model's ontology.

Model parsimony links strongly with the data we put into our simulations and the data we validate them against. Due to the inherent limita-

tions of the archaeological record, it may not be immediately obvious what kind of data would be generated as a result of a known phenomenon, if any. Frequently, we hypothesize a general process that might have driven the dynamics of past societies that, in the absence of writing, is only theoretically plausible and might have left behind only circumstantial archaeological evidence.

In these situations, we would want to develop a model closer to the theory-driven end of the spectrum as it will enable us to make leading steps toward identifying what kind of data or data pattern we should look for. These are often built up from first principles and are frequently based on theories developed in other disciplines. Since archaeological inquiry concerns virtually all aspects of the past, the pool of base models we can draw from extends over all scientific disciplines concerning human groups—demography, economics, sociology, political science, epidemiology, and others. Thus, it is a good strategy to pick up an existing base model from one of these related scientific fields, critically adapt it to your research context, and then examine how it would play out in our models of the past. The *in silico* archaeological record it generates in the process can then be compared against the physical archaeological record to establish whether the baseline scenario we just tested could be offered as a feasible explanation.

Chapters 4–6 provide an overview of established theoretical frameworks and well-established models.

Often, the key aspects of past systems we are interested in studying are relationships. These relationships may manifest in similarities between assemblages at different sites, ceramic decorations, trends in architecture, presence of exotic goods, or written mentions. In all cases, data that include relationships need to be treated with robust methods developed to deal with their peculiarities. One of the frameworks used for relational data is **network science**. By representing relationships in the form of a network, we can dramatically simplify our models and make them more robust.

network science: branch of science concerned with relational data and its collection, management, analysis, interpretation, and presentation (Brandes et al. 2013).

In this chapter, we will use a theoretical model from Crabtree (2015) to demonstrate how to incorporate general theories of exchange into a simple NetLogo model. We will also follow and expand on a tutorial for calculating network statistics by Brughmans (2016) to demonstrate how to use the network extension in NetLogo.

8.1 Theories on Exchange

In 1972, Sahlins published his theories on economics in small-scale societies. They have been recently embraced by many anthropologists attempting to understand how nonmonetary societies establish rules for survival (Crabtree 2015; Hegmon 1989; Kobti 2012). Each of these specifically model exchange between individuals via a relational network rather than just spatial proximity.

Anthropological theorists differentiate between **exchange** and **trade** (Sahlins 1972). Exchange is a broader term with diverse interpretations that incorporates all kinds of exchanged goods, services, and ideas. The term *trade* is often used as a narrower version of exchange, frequently involving money as an intermediary and in a market-based context (Renfrew and Bahn 2017, 374–376). Monetary-based transactions can ensure that an individual (or group of individuals) gains income that they then can use to purchase other objects; nonmonetary-based exchange is generally related more to necessities and thus better reflects relationships and group cohesion. In the absence of direct observation of how small-scale societies in the past functioned, these theoretical models of exchange provide a basis for testing how exchange could lead to societal growth, ensure greater survival during years of hardship, or enable group cohesion. Sahlins (1972) defined four types of nonmarket-based exchange:

exchange: the giving of one object and receiving of another, usually in the same type, and especially goods, services, or ideas.

trade: exchange that involves money as intermediary and is market-based.

- pooling of resources;
- generalized reciprocal exchange network;
- balanced reciprocal exchange network; and
- negative reciprocity.

Pooling refers to when an individual shares their resources among all participants. In households, there will often be more individuals present than there are providers, so resources may be *pooled* to a head of household and redistributed equally to household members. Pooling generally comes with no expectation for immediate reimbursement but considers long-term implications. For example, while we may not expect a four-year-old to supply for the family, when the child is grown she may be expected to take care of her parents. Pooling is always reserved for very close kin, and is generally kept at small groups of people as it is diffi-

pooling: type of exchange where resources are shared among all.

cult to scale up, though some societies do pool resources even for larger camps (Bird and Bird 2008).

generalized reciprocal exchange networks (GRN): type of exchange where individuals share their surplus with their kin when asked.

Generalized reciprocal exchange networks (GRN) extend beyond the limited sphere of the in-group that pooling refers to. In generalized reciprocal exchange, an individual will share with relatives if the individual has surplus. Unlike pooling, where sharing is always assumed, in GRN the relative will only ask when they fall short. However, assistance will always be given if there is surplus (albeit sometimes grudgingly), with the idea that the giving individual can always rely on others if they fall short. Sahlins suggests that this will only occur with closely related individuals—siblings, parents, children, first cousins, uncles, and aunts. Otherwise, exchange falls into the next category.

balanced reciprocal exchange networks (BRN): type of exchange where individuals may lend to others but expect to be paid back by the individual who borrowed from them.

Balanced reciprocal exchange networks (BRN) link any individual to any other individual, but generally follow a reputation network. If an individual falls short and needs to borrow, they will be expected to pay back in kind when requested by the lender. If they are unable to pay back, their standing drops, and the borrower will be unlikely to be able to borrow in the future. If individuals share information, you can imagine that the reputation of the borrower would suffer more widely, incentivizing the borrower to repay their loans.

negative reciprocity: a type of exchange where an individual takes advantage of another without any intention to repay the debt.

The final type of exchange, **negative reciprocity**, refers to trying to receive something without having to give back for it. It can refer to things like raids on other groups, or tricking an individual into giving something for nothing, thus always benefiting the recipient at the detriment of the giver.

In subsistence-based societies with nonmarket exchange, these types of exchange networks appear to be the backbone to how societies function. Thus, they provide a useful theoretical platform for modeling how exchange networks could have worked in the past, how they may provoke growth and increase resilience, and what the underlying social structure may be when one type of exchange is more prevalent over another.

Building these rules into a simple model, we will work through the NetLogo Network extension to create a theory-driven model, similar to the model presented in Crabtree (2015). We will model pooling, GRN, and BRN to demonstrate how these three types of networks can interact together or in various combinations.

8.2 Sahlins's Model of Exchange

We will start modeling the pooling algorithm by creating multiple breeds representing four different households who can then exchange goods—here we will exchange pots.[1] Open NetLogo and a new blank model. Create four breeds, named `lineageA` through `lineageD`, at the start of the model. As with the GIS extension in chapter 7, we will be using an extension called `nw` (short for Network) instantiated at the top of our code.[2] We also need to create `turtles-own` variables (which apply to all breeds), that is, a `head` of household variable to know who to pool resources toward and a `pots` variable to keep track of the resources. Also, give turtles variables called `BRN-list` and `reputation`, that will be needed later.

Extensions are an easy way to boost NetLogo functionality.

Model:
Sahlins's Model of Exchange by Crabtree

ABMA Code Repo:
ch8_networks

Next, we create our `setup` procedure, building in our lineages.

CODE BLOCK 8.0

```
to setup
  clear-all
  create-lineageAs 1 [create-households "A"]
  create-lineageBs 1 [create-households "B"]
  create-lineageCs 1 [create-households "C"]
  create-lineageDs 1 [create-households "D"]

  reset-ticks
end
to create-households [name]
    set label (word "head of household" name)
    set head who
    setxy random-xcor random-ycor
    set pots 1
    set color blue
    set reputation 0.95
    set brn-list (list)
```

DON'T FORGET
When initializing breeds, NetLogo requires both a plural and a singular name.

TIP
Here we pass an input `name` to the procedure `create-households`, in our case, this is the *name* of each lineage: "A," "B," etc.

[1] You can find all code written in this chapter in the ABMA Code Repo: https://github.com/SantaFeInstitute/ABMA/tree/master/ch8

[2] Find the full extension list, including complete documentation for the Network extension, on NetLogo's website: https://ccl.northwestern.edu/netlogo/docs/nw.html.

CODE BLOCK 8.0 (cont.)

```
    let N random 15 + 1
    hatch N [
      rt random-float 360 fd 1
      set label (word "lineage" name "s")
      set pots random 5
  ]
end
```

Here we create one individual, label them as head of household, and then place them randomly in the landscape. We then choose a random number between 0 and 15 and `hatch` that many additional individuals around the head of household; note the `+ 1`, which ensures that the family is always 2 or more for pooling purposes. This creates a random number of individuals the head will have to "provide" for. Repeat this procedure for lineageB, lineageC, and lineageD so that we can have four families in our society.

DON'T FORGET
Create a switch called `pooling?` on the INTERFACE.

The following `go` procedure includes everything we will write in this chapter, though we will leave several procedures commented out for now (note the semi-colons):

CODE BLOCK 8.1

Alternatively you can write the procedure calls `to repay-BRN ... end` without anything inside. NetLogo will just pass over them without triggering the debugger.

```
to go
  produce-pots
  ;repay-BRN
  ;reputation-update
  if pooling? [ pool-resources ]
  ;if GRN? [ GRN-exchange ]
  ;if BRN? [ BRN-exchange ]
  consume
  ask turtles [set size pots]
  tick
end
```

Agents will produce and consume pots, and exchange them by either pooling or engaging in generalized reciprocal exchange (GRN) or balanced reciprocal exchange (BRN). We can imagine that maize or any other food

substance would be traded within pots, but since pottery is a durable archaeological good, we use these as proxy for the traded substance. To help our visualization of the flow of goods, at the end of `go`, we ask the agents to adjust their size based on their current pot count.

We will model changing numbers of produced and consumed resources because exchange is influenced by scarcity and abundance, so having heterogeneity in the number of objects per individual in the group is necessary:

CODE BLOCK 8.2

```
to produce-pots
  ask turtles [
    let M random 3
    set pots pots + M
  ]
end
```

The consumption of pots can stand in for consumption of the resource that was inside, natural breakage of ceramics, or just attrition, and will act to reduce the ability of individuals or households to stockpile ceramics.

CODE BLOCK 8.3

```
to consume
  ask turtles [
    let B random pots
    set pots pots -  B
  ]
end
```

To write our first exchange procedure, pooling, we begin by asking the agents within a lineage to produce pots, and then give all of those pots to their head. The head of the household will then redistribute pots to each individual in their lineage, ensuring an even distribution.

CODE BLOCK 8.4

```
to pool-resources
  ask turtles with [head != who] [
    ask turtle head [set pots pots + [pots] of myself]
    set pots 0
  ]
```

CODE BLOCK 8.4 (cont.)

```
  ask turtles with [head = who] [
    let my_turtles turtles with [head = [who] of
    myself]
    let share floor (pots / count my_turtles)
    while [pots > share] [
      ask min-one-of my_turtles [pots] [
        set pots pots + 1
      ]
      set pots pots - 1
    ]
  ]
end
```

The rounding primitives, `floor` and `ceiling`, ensure that you don't divide pots or humans into fractions. In some cases, it requires additional code to deal with the remainder.

Here the head of the household is given all the pots, and then iterates through their family group, giving a pot to each member, until all pots are distributed. This means that sometimes some individuals in the household will receive fewer pots, but this will be random. Set up and run the model at this point and see how the agents' sizes, representing their pot count, fluctuates with each time step.

8.3 Network Models

network: a set of entities (nodes) connected through a set of relationships (edges or links) that represent any type of relationship, from kinship and friendship to trade flows and material culture similarity.

topology (in ABM): the relative organization of space/geometry in which agents operate. This may be a 2D plane, cylinder, torus, or network.

Network science has become an increasingly common approach in archaeology (Brughmans 2010), thanks to the multitude of applications and relatively simple analytical methods. In ABM, **networks** provide a **topology** alternative to the standard spatial approach in cases where relationships matter more than the spatial location. A good example is a peer network—your friends may live on the other side of town, yet you interact with them more often than some of your neighbors living in the same apartment block. In those cases, a network is a more appropriate representation of your relationships.

The two components of a network, often referred to as a graph, are **nodes** and **edges**. Nodes are the individual entities represented (e.g., individuals, groups, towns), while edges are the connections between them (e.g., friendships, alliances, roads). Edges can be directed or undirected, depend-

ing on whether their relationship is symmetrical. If two archaeological assemblages A and B are linked because of similarity, that similarity is the same, whether looking from A to B or B to A, so an undirected graph is used. On the other hand, directed edges can be used to represent a one-way link, such as a one-way street or a link representing feeding behavior (e.g., a forager eats a plant). Bidirectional edges mean that the link between the two nodes is reciprocated, such as a two-way street, mutual friendships, or an exchange relationship.

nodes: the entities in a network (the dots).

edges: the relationships between the entities in a network (the lines).

What the edges and nodes represent, their direction, and how the connections are established depends on your research questions. This could be informed by empirical research, simulated according to a specific process such as preferential attachment, or formed at random. Here we will model nodes as individual people and edges as the exchange relationships between them.

Network science comes with a wealth of tools to generate networks with different structural properties. There are several network types formed through a randomized process that are often used as null hypotheses (Brughmans 2016). Here we will use the **Erdős-Rényi** random graph model (Erdős and Rényi 1960). In an Erdős-Rényi network, we randomly choose two nodes from all possible nodes and connect them as a pair; this process is repeated until the model reaches the number of edges defined by the user, forming a network (Newman 2010; Brughmans 2016). We use random networks in several circumstances: when the exact connection doesn't matter (e.g., because the agents are initially identical), when we don't know anything about the modeled network so the best bet is to model it as a random one, or when we're interested in how different social networks would perform and need a baseline. In our model, we examine an Erdős-Rényi network to examine whether observed networks differ from randomly created networks (the null hypothesis). We will connect pairs of agents but limit the edges to the out group, meaning that we will not connect two nodes of the same lineage, though we do allow nodes to have multiple edges.

Erdős-Rényi network: a network, also known as a graph in network science, created by repeatedly connecting two random nodes with an edge until the specified number of edges is reached.

In NetLogo, edges are called `links`, and they can be used in a similar way to turtles (Brughmans 2016). We will name our links' breed `edges` to be consistent with network science lingo. At the top of your code, below `turtles-own`, add:

CODE BLOCK 8.5

```
undirected-link-breed [edges edge]
```

Next we will create the generalized reciprocal exchange network, `GRN-setup`, called from the `setup` procedure to initialize the network.

CODE BLOCK 8.6

```
to GRN-setup
  let Aconnections count turtles with
  [ breed = lineageAs ] * count turtles with
  [ breed != lineageAs ]
  let Bconnections count turtles with
  [ breed = lineageBs ] * count turtles with
  [ breed != lineageBs ]
  let Cconnections count turtles with
  [ breed = lineageCs ] * count turtles with
  [ breed != lineageCs ]
  let Dconnections count turtles with
  [ breed = lineageDs ] * count turtles with
  [ breed != lineageDs ]
  let totalLinks (Aconnections + Bconnections +
  Cconnections + Dconnections)

  repeat (target-density * totalLinks  / 2)[
    ask one-of turtles with [count edge-neighbors <
    count turtles with [breed != [breed]
    of myself]] [
      create-edge-with one-of other
      turtles with [breed
      != [breed] of myself and edge-with
      myself = nobody]
    ]
  ]
end
```

Try creating a list of lineages `["A" "B"...]` and use `foreach` to sum up all of the possible links rather than counting each lineage's links in turn before adding them together.

Because we randomly create the number of individuals in each of the lineages, we must calculate the total number of connections each lineage

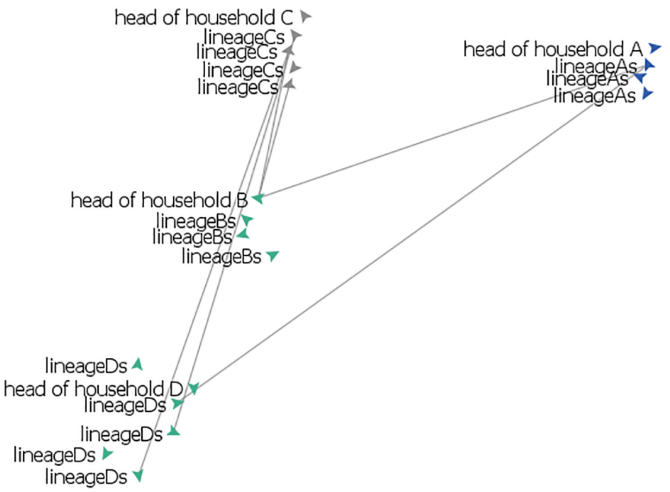


Figure 8.1. Default network layout in NetLogo. Members of each of the four lineages cluster around the head of household.

could possibly have (`let Aconnections...`). We then sum each of these numbers together to get the total possible number of links and divide by two because edges are **undirected**. Since we want to control the density of the network,

we create a slider to represent the proportion of this maximum number of edges, TARGET-DENSITY: 0 represents no edges, 0.5 represents half the maximum number of edges, and 1 represents all possible edges.

We then loop over a process where a random agent creates a link to someone outside their household, thereby building up a generalized reciprocal exchange network link by link. According to Sahlins, these would be distant kinship ties, so what we are doing is creating a network of individuals who may count themselves as somewhat related and be willing to offer assistance when called upon, without expectation for direct repayment. Add `GRN-setup` at the end of your setup procedure after you have populated all your lineages (fig. 8.1).

One aspect of network models is their ability to quickly communicate the dynamics of a system using visualizations. NetLogo includes a few `layout-` primitives for helping to rearrange the network. Some, like `layout-spring` , must be run many times using `repeat` before they settle into a stable configuration. Here we will just use the simpler `layout-`

directed edges: one-way connection, such as a predator eating a prey (trophic networks).

undirected edges: two-way connection, such as a reciprocal relationship.

DON'T FORGET
Add a slider, TARGET-DENSITY, to the INTERFACE tab, with values 0–1 and increment of 0.05.

TIP
Turn off horizontal and vertical wrapping using the SETTINGS button on the INTERFACE to improve the visualization.

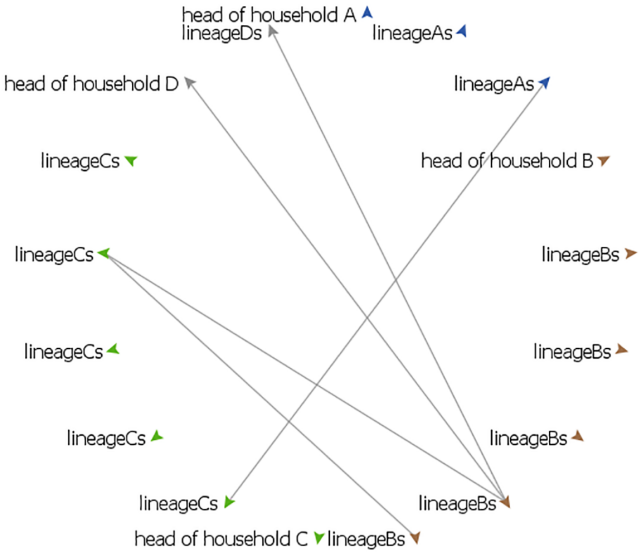


Figure 8.2. Rearranging a network to be in a circular layout. Here the target-density is lower than in the previous figure, resulting in fewer links overall.

`circle` to help visualize the link density, though we will keep the lineages together by sorting the turtles according to their `who` number. Near the end of `setup`, after `GRN-setup`, add:

CODE BLOCK 8.7

```
layout-circle sort turtles 10
```

Changing a network's layout doesn't change its structure. It's only a way of visualizing it. The process we will run on the network is unaffected by the layout.

The `sort turtles 10` sets the order and spacing of the nodes in our network. If you change the number to something smaller, like 5, you will note the network has a smaller radius. If you leave out `sort`, the agents will be in a randomized order and not near their lineage members. Adjust `target-density` and hit `setup` a few times. You'll see how the network changes with greater and lesser density of edges.

`sort turtles` according to their `who` works because of the order we created them.

Now that the network is arranged, we can code the actual generalized reciprocal exchange algorithm. Write the following code, add a `GRN?` switch to the INTERFACE, and uncomment the `GRN-exchange` line from the `go` procedure:

CODE BLOCK 8.8

```
to GRN-exchange
  if any? turtles with [pots < 2 and any?
  edge-neighbors with [pots > 0]] [
    ask one-of turtles with [pots < 2 and any?
    edge-neighbors with [pots > 0]] [
      if random-float 1 < exchange-probability [
        set pots pots + 1
        ask one-of edge-neighbors with [pots > 0]
        [set pots pots - 1]
      ]
    ]
  ]
end
```

In this piece of code, agents request a pot from one of their connections and exchanges given a set probability. If the counterpart has any pots left, the exchange takes place. This is a simple way for agents to create a network and exchange resources within that network. You can expand it by, for example, assessing biases in choosing an exchange partner or by analyzing economic mechanisms of when to exchange and when not to.

For the code to work, you need to return to the INTERFACE tab and create an `exchange-probability` slider, set between 0 and 1, with increments of 0.05.

For more algorithms relevant to different types of exchange, see chapter 5.

In network science, many metrics explore the position and connectedness of a node within the network, measuring, for example, how central a node is in comparison to its neighbors. While we tend to use R or Python to analyze our model results (see ch. 9), NetLogo does have built-in abilities to perform simple network analyses using the Network extension. This allows you to calculate network measures on the fly and spot-check the model's functionality before running an entire experiment. We will use a reporter to calculate the **average degree** of the network and then display it in the INTERFACE tab (fig. 8.3). Average degree measures how many connections an individual node has on average (Newman 2010; Brughmans 2016). For example, a network of 10 nodes and 20 edges will have an average degree of 2 if it is a directed network, or an average degree of 4 if it is bidirectional.

average degree (networks): the average number of edges per node in a network.

Here we follow Brughmans's (2016) calculations for reporting average degree in a NetLogo model. Add the following reporter to your code:

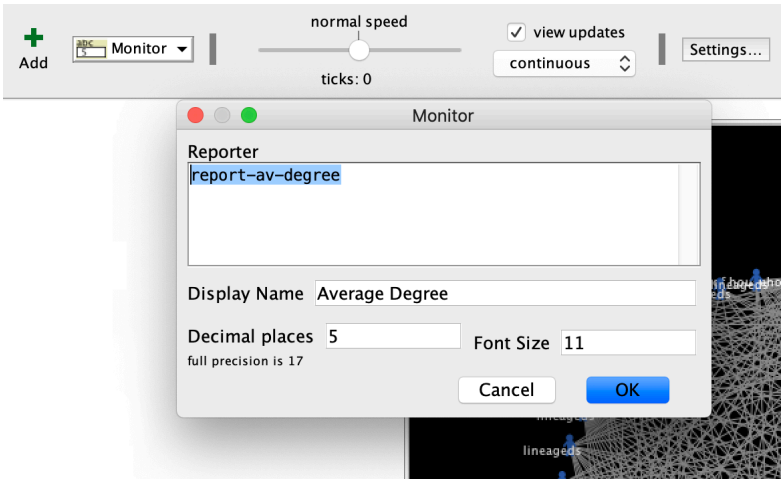


Figure 8.3. Adding a REPORTER as a monitor to show the average degree of the network.

CODE BLOCK 8.9

```
to-report report-av-degree
  let av-degree sum([count edge-neighbors] of turtles)
  / (count turtles)
  report av-degree
end
```

We can then display average degree in the INTERFACE tab. Return to the INTERFACE and add a MONITOR. Write `report-av-degree` in the REPORTER box, and call it `Average Degree` in the DISPLAY NAME box. Change the number in the DECIMAL PLACES box from the default to 2, and click OK. Now every time you hit SETUP, you'll see the average degree of the network. If you change `target-density`, the average degree will change (fig. 8.3).

TIP
There are many types of statistical measures in network science. We recommend Mark Newman's book *Networks: An Introduction*.

The NetLogo `nw` extension is extremely useful for running other network statistics without the need to calculate them by hand as we do above for average degree. We have already added the network extension at the top of the code, so to freely use its functionality we only need to tell it what are the nodes and what are the edges. At the end of your setup procedure, type:

Note that primitives from the extension start with `nw:`

CODE BLOCK 8.10

```
nw:set-context turtles edges
```

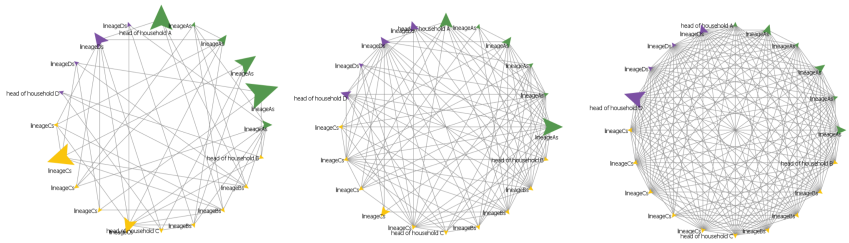


Figure 8.4. Three example networks showing average degrees of 5, 10, and 16 (close to maximum). Note that the density increases nonlinearly.

Here, we tell the `nw` extension that `turtles` are nodes (first context) and `edges` are edges (second context). If you only wanted to calculate the network of a specific breed, you would input that for the first context. All is set up now to calculate the **average shortest path length** of the network.

average shortest path length (networks): the mean number of steps between all pairs of nodes in a network.

Average shortest path length measures how many steps, on average, it takes to get from any one node to any other node using the shortest route (Newman 2010). Networks with low average shortest path length are known as small-world networks, because it is easy to get from any one node to any other node. Type at the very end of your code:

CODE BLOCK 8.11

```
to-report average-shortest-path-length
  report nw:mean-path-length
end
```

Back on the INTERFACE tab, add another MONITOR, and in the REPORTER box enter `average-shortest-path-length`. The average shortest path length reporter will not work if any of your nodes are isolated (i.e., have no trading partners) as the metric cannot be calculated and your monitor will report `false`.

If you repeatedly get "false," it's likely that there aren't enough connections for all of the nodes in the network to be linked in. Increase your `target-density` to make more connections.

Finally, let's explore the **betweenness centrality** of the nodes. In network analysis, betweenness centrality measures the number of shortest paths that cross a node; at the graph level, it averages these measurements for the whole network. It's a useful measure for examining how well the network is connected. If measured for a single node, it shows its position with respect to the rest of the network.

betweenness centrality of a node (networks): the number of shortest paths between all pairs of nodes that cross through the node.

Add another monitor to the INTERFACE, name it "betweenness" in the DISPLAY NAME, and write in the REPORTER box:

CODE BLOCK 8.12

```
mean [nw:betweenness-centrality] of turtles
```

betweenness centrality of a network (networks): the average betweenness centrality of all nodes of a network.

Now your network model will display the average betweenness of the network. Try different `target-density` values and see how they impact the three network measures we have monitors for. Feel free to explore the other types of network measures available from the `nw` extension manual and follow the instructions there to examine outputs like eigenvector centrality, clustering coefficients, and other measures.

While monitors are useful for seeing summary statistics at the network level, plots are useful for examining their full distribution. To better understand the difference we will create a plot of the degree distribution across the agents in our model.

In the INTERFACE tab, create a new plot and name it "Degree distribution," change it to a BAR plot, and write the following code in the PEN UPDATE COMMANDS box (fig. 8.5):

CODE BLOCK 8.13

```
let max-degree max [count edge-neighbors] of turtles
set-plot-x-range 0 (max-degree + 1)
histogram [count edge-neighbors] of turtles
```

The x-axis of the histogram shows the degree, while the y-axis shows the number of nodes that have that degree.

You should now see a histogram displaying the frequency of the agents' degrees (the number of edges) in their generalized reciprocal exchange network. Note that this plot will only change on setup, since we do not allow for adding or subtracting edges in the GRN. This will change when we introduce the BRN exchange.

Finally, create another plot to track the distribution of pots. This plot will be much more dynamic, since at every time step the number of pots an individual has will change as a result of pooling, generalized reciprocal exchange, discard/breakage, etc. As above, add a new plot, name it "Number of pots," and change its mode to a BAR plot (histogram). Use the following code in the PEN UPDATE COMMANDS box:

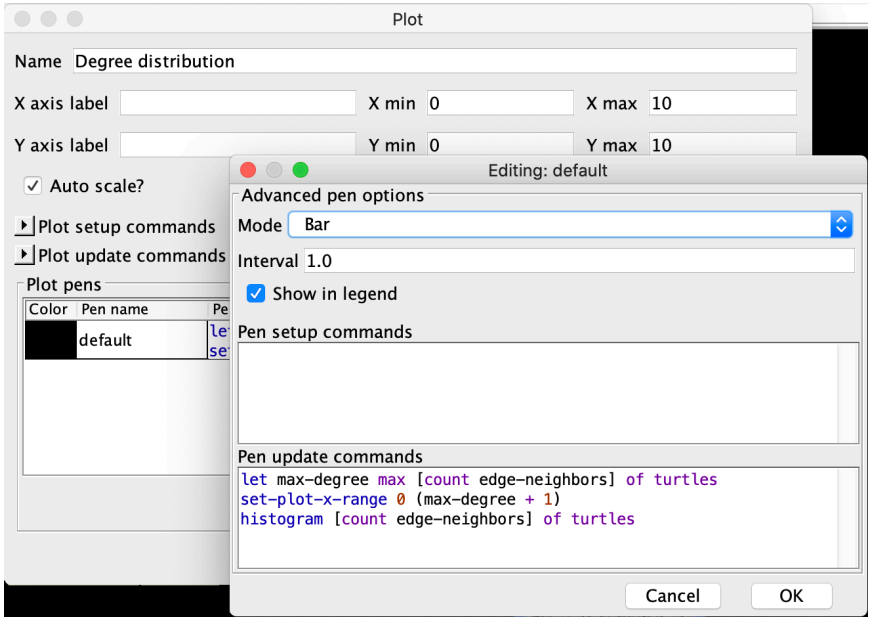


Figure 8.5. Creating a customized plot for degree distribution of the entire network.

CODE BLOCK 8.14

```
let max-pots max [pots] of turtles
set-plot-x-range 0 (max-pots + 1)
histogram [pots] of turtles
```

Run the model a few times to see how the histograms change. If you modify the `target-density` of your network, the degree distribution and the number of pots will change accordingly.

Finally, we will code one more exchange type identified in Sahlins's theories of exchange: the balanced reciprocal exchange network (BRN), which is based on reputation and requires in-kind payback. We will simplify it by assigning a reputation to agents that will decrease if they fail to pay back their debts. In reality, reputations may differ depending upon who is assessing them, but for this model reputation will be held constant and visible for all agents. Write a new procedure using the following BRN exchange algorithm:

CODE BLOCK 8.15

```
to BRN-exchange
  if any? turtles with [pots < 2] [
    ask turtles with [pots < 2] [
      if random-float 1 < reputation and random-float
      1 < exchange-probability-BRN [
        let lender one-of turtles with [pots >= 2]
        ask lender [ set pots pots - 1 ]
        set pots pots + 1
        set BRN-list lput [who] of lender BRN-list
        create-edge-with lender
      ]
    ]
  ]
end
```

DON'T FORGET
Add another 0-to-1 slider for `exchange-probability-BRN`.

The above code follows a similar logic to the GRN algorithm. An agent that is running out of pots has a probability of assessing whether their reputation and is sufficient to ask for a loan. If it is, that agent then asks another agent for a pot. The lender's `who` number is added to the list so that the agent can remember who to return pots to. You will note that the requesting agent can ask anyone in the world to exchange with; this roughly follows Sahlins's logic that BRN exchanges will be with anyone, regardless of their relation to you. Finally, we create an edge with the lender to update our social network.

DON'T FORGET
Remember to uncomment `BRN-exchange` from `go` and add a switch for `BRN?` on the INTERFACE.

The global reputation of an agent will depend on how often they borrow, and whether they pay back the lender. Write a procedure that updates the reputation:

CODE BLOCK 8.16

```
to reputation-update
  ask turtles [
    ifelse length BRN-list > 0 [
      let R ( 0.05 * length BRN-list )
```

CODE BLOCK 8.16 (cont.)

```
      if reputation - R > 0 [
        set reputation reputation - R
      ]
    ]
    [ set reputation 0.95]
  ]
end
```

In the above code, an agent sets their reputation to 0.95 if they do not currently owe pots to anyone; it is not quite 1 to enable a small probability that a BRN exchange would be turned down. For every entry on the list, we decrease reputation by 5%, making it so that agents will be penalized for borrowing too frequently.

Notice that agents' reputation cannot fall below 0.

Finally, to make this exchange "balanced," the agents need to have a method for paying back their lenders. Add another procedure:

CODE BLOCK 8.17

```
to repay-BRN
  ask turtles [
    if length BRN-list >= 1 [
      while [length BRN-list >= 1 and pots > 1] [
        set pots pots - 1
        let lender first BRN-list
        ask turtle lender [ set pots pots + 1 ]
      ]
    ]
  ]
end
```

In this algorithm an agent assesses their list of debts and, as long as they have more than one pot that they iterate through their list, repays their lenders. The interaction between this and the reputation algorithm means that once a debt is paid off, reputation will be reassessed and set to reflect the current number of debts.

When running this model multiple times, you can see how some individuals end up with more pots while others end up with fewer pots, leading

to an increase in inequality over time. Now imagine that death was tied to whether or not you had any pots, and that children of agents inherited their network connections. In her model, Crabtree (2015) explored how that would impact agents' survival. She demonstrated that pooling within a household, GRN among relatives, and BRN tied to reputation in a wider community increased agents' survival chances by buffering them from catastrophic events such as droughts. Although this is a largely theory-driven model, by comparing the simulated populations to the real archaeological populations, Crabtree demonstrated that the exchange networks theorized by Sahlins would have structured the real Ancestral Puebloan society.

Run the model with different combinations of exchange strategies (pooling, GRN, BRN) switched on. How do the network measures change?

The applications of both network science and ABM in archaeology are promising and the number of publications is slowly growing (Brughmans 2013). Crabtree (2015) built a more complex version of the above model to examine how exchanges impact survival for the Ancestral Pueblo of southwestern Colorado in the *Village Ecodynamics Project* agent-based model. Brughmans and Poblome (2016) represented the eastern Roman Empire as a network of markets; Graham (2006) used the Antonine Itineraries as a baseline transport network for modeling the spread of information; Graham (2009) developed a model of interpersonal relationships in ancient Rome based on prestige and gift exchange; and Crabtree (2016) used networks and simulation to examine the beginnings of the wine industry in southern France. In more abstract frameworks, Cegielski and Rogers (2016) used networks to model how individual behavior leads to formation of chiefdoms, and White (2013) combined networks with cultural transmission. Network approaches combined with agent-based modeling provide a useful method for examining the many different ways in which past peoples were connected and to further refine our understanding of how relationships shaped their lives.

8.4 Code Testing

You have now written quite a few agent-based models and undoubtedly have come across many code errors in the process. At the early stages of acquiring coding skills, the majority of problems come from incorrect syntax, such as missing brackets or wrong data types being passed (e.g., a primitive asks for an agentset and is given a number instead). These kinds of errors,

even if frustrating, are easy to catch thanks to NetLogo's built-in debugger and will dramatically drop in frequency after only a few months of coding practice.

Much more difficult to catch are errors in the logic of the code. They represent a situation in which there is a discrepancy between what the code does and what the modeler's intentions are. In most cases, they will not be caught by a debugger, so a good testing strategy is necessary to ensure that the results we present are not an artifact of buggy code. Here we will describe a testing workflow that combines a few of the techniques that can be used to find and remove bugs from your models.

NetLogo has a suite of options for observing agents and patches. Right-click on a turtle and you will find INSPECT, WATCH, and FOLLOW in the drop-down menu.

VISUALIZATION

The strategy we've been deploying all along is simple visualization. If your agents do not move when they should move, then you know there is something wrong with the code. Similarly, you can spot-check whether their behavior is correct by inspecting a few randomly chosen agents over the course of the simulation. Throughout the book, we have done quite a lot of reporting onto the INTERFACE using monitors and plots and inspecting the variables of individual turtles by right-clicking on them. These are exceedingly useful for checking that the dynamics of the model makes sense. And in many cases you can use the COMMAND CENTER to spot-check the simulation. For example, you can write `mean [reputation] of turtles` a few times during the execution of the model to see whether its value is within a reasonable range. Often we build plots just as a form of testing, and then remove them when finalizing the project. For example, in our case it may be worth plotting the pots distribution for each lineage separately to ensure that each lineage has pots, something you may miss if looking at all the turtles at once.

Copy-pasting code segments is an extremely common source of bugs.

PRINT STATEMENTS & MANUAL TESTS

Even if everything looks fine, that is not a guarantee of error-free code. As we develop the code, we need to go through each algorithm step by step to ensure that it does what it is supposed to do. This is the strategy we discussed in chapter 2: armed with pseudocode, follow the logic of each procedure and insert print statements when necessary. For example, if we want to as-

sure that a given procedure is called the expected number of times, we can check it this way:

CODE BLOCK 8.18

```
to repay-BRN
  print count turtles
  print count turtles with [length BRN-list = 0]
  ask turtles [
    if length BRN-list >= 1 [
      print "Turtles with debts"
      print "1"
      while [length BRN-list >= 1 and pots > 1]
      [
        print "repaying is happening!"
        set pots pots - 1
        let lender first BRN-list
        ask turtle lender [ set pots pots + 1 ]
      ]
    ]
  ]
end
```

The first two print statements will give you the number of turtles that should engage in the `repay-BRN procedure` (total number minus turtles with no debts). If it's the same as the number of "1"s printed to the COMMAND CENTER, then your code is proceeding the right way. In addition, we check that the repaying is actually happening by inserting another print statement into the repayment code block. It is important to do that after all conditionals to ensure that there are in fact turtles that fulfill the condition. If the COMMAND CENTER does not print the "repaying is happening!" statement, we will need to investigate why it is that no agents have enough pots to pay off their debts.

TIP
It's easier to perform this type of test if you limit the number of agents.

Similarly you can check that the procedure is performing correctly by checking its status and all the variables involved before and after the execution.

```
to repay-BRN
  ask turtles [
    print brn-list
    if length BRN-list >= 1 [
      while [length BRN-list >= 1 and pots > 1][
        print "-----------"
        print pots
        set pots pots - 1
        print pots
        let lender first BRN-list
        print [pots] of turtle lender
        ask turtle lender [ set pots pots + 1 ]
        print [pots] of turtle lender
      ]
    ]
  ]
end
```

CODE BLOCK 8.19

You can also test the "accounting" by checking that the total sum of the pots before and after the transaction remains the same.

Here we check that the number of pots before and after the exchange is correct. The examples given here may look simplistic, but the whole point of a bug is that you aren't aware of it, so testing even the simplest of procedures is necessary. Don't forget to remove the print statements after testing, as they consume a lot of computational power.

TIP
You might have noticed the `print "-----------"` line. This is a handy way to visually separate each iteration of the code.

EXTREME SCENARIO TESTING

The previous testing strategies are performed during code development. Once the model is written, we can focus on its general workings. For example, it is worth running the simulation with values that are outside of the normal range to probe the robustness of the model's logic. What happens if there are only two agents? Or just one? Or none? What if we set the `target-density` to 0 or to 1—does the model still run? What happens if an agent doesn't have anyone to trade with or has two equally good partners to choose from? When running extreme scenarios, it is a good practice to test what happens if values are set to 0, 1, a negative number, and a very

large number. It usually provides insight into the behavior of the model in situations that may be rare but still possible.[3]

ASSERTIVE TESTS

As you develop your code, the interactions between procedures become increasingly complex, making it difficult to identify problems. With each new procedure, we may be introducing unintentional behavior into procedures that were developed and tested long before. Building assertive tests into the code is one of the best ways to maintain control.

For example, instead of printing the number of pots of each agent, as we did in the previous section, you can test whether the total number of pots before and after exchange is the same.

CODE BLOCK 8.20

```
to go
  ...
  let sumPots sum [pots] of turtles
  repay-BRN

  if sumPots != sum [pots] of turtles [
    print "We've lost some pots in exchange!"
    type sumPots type " " print sum [pots] of turtles
  ]
  ...
```

The primitive `type` is like `print` but does not make a new line so you can write several variables and strings together. Also check the `show` and `word` primitives.

Assertive statements will alert you to when a problem appears only in a very specific situation, that is, the model runs fine except when a rare circumstance happens. These are some of the most difficult bugs to catch since they are unlikely to pop out when you do spot checks. Moreover, these situations may occur when you run a large parameter sweep of your model, aborting a run that may be critical for analyses (see ch. 9). You can also use assertive statements for a so-called reality check. Generating numbers describing the systems as a whole (e.g., total number of exchanged pots) can

Another simple yet powerful testing technique is to ask a colleague to do a code review of your model.

[3] It is extremely rare but nevertheless possible that there's a bug in the NetLogo base code. Testing extreme case scenarios once landed one of the authors on a path to a bug in a core Python library while another author found an issue in NetLogo's `in-radius` primitive. In those cases, you need to submit an issue to the developers so that they can patch the code: https://ccl.northwestern.edu/netlogo/help.shtml#reporting-bugs.

help us to ensure that the artificial world we have created does not exhibit idiosyncrasies. These may not necessarily be code errors but rather unintentional behaviors of the model. If four families of about ten people each exchange 5,000 pots over one tick, we know that this is not a plausible model. Assertive statements help in catching such cases.

Finally, a bit of calculator mathematics is needed to check that the probabilities that we have built into the model are consistent with our expectations. For example, in a model including demography, you can print the number of new babies versus the total number of turtles to ensure that your 5% probability of producing an offspring is at approximately the correct probability. Probabilities have a tendency of becoming increasingly dynamic if numerous conditions need to be met. For example, if on top of your 5% probability of producing offspring, agents also need to be in spatial proximity to another agent, then these two probabilities interact. When the population is small, the probability of being in close proximity to another agent is low, but 200 time steps later, when the number of agents is much higher, the probability will increase. This means that the actual probability of producing an offspring changes over the course of one run. Sometimes this is fine or irrelevant for the results; sometimes it is not. Either way you need to identify and control for these kinds of dynamics.

These are only a few debugging techniques, so we strongly recommend exploring this topic further to learn about more robust methods such as test-driven development and unit testing.

Ultimately, the best (while time-consuming) method of assuring the correctness of a model's results is **replication** by another researcher (Edmonds and Hales 2003). If done independently, this has a high potential of finding even the most hidden flaws in logic or hidden code bugs. It is also worth spending some time replicating colleagues' models to ensure that the self-correcting nature of scientific inquiry is maintained. Designing, running, and interpreting a simulation are some of the hardest tasks in research, and because of the formal nature of the work, they are much more exposed to scrutiny than many other types of research. Transparency is the best way to ensure the integrity of our work, even if it leads to inevitable "oops" moments. Digital archaeologists can be harsh critics, so it is worth keeping some perspective on one's own and others' work. Plenty of the current archaeological models may turn out to be flawed in one way or another, but they are nevertheless useful in pushing our understanding forward.

replication: a reimplementation of a model by another researcher to verify the original model's correctness.

8.5 Summary

Building agent-based models up from theoretical frameworks developed in adjacent disciplines can help us refine our understanding of complex phenomena. Even in the archaeological record, where we may not be able to observe social processes directly, models that incorporate established theories can aid in helping refine our understanding of the past. Further, by formally modeling these theories, we can test them and advance our theoretical understanding. This is an important consideration when tackling the complexity of many social and natural systems. Simple explanations do not hold water for most systems in the present, so it is equally unlikely that they would do so for the past. Deploying formal methods enables us to start from simple explanations and then continue building on them, getting closer and closer to the complexity of peoples' lives 200, 2,000 or 20,000 years ago.

We also briefly introduced the field of network science. By using the `nw` extension and established mathematical principles of network statistics, we can evaluate how the structure of social (and other types of) networks might have impacted past communities. These techniques enable us to simplify complex phenomena in a well-developed and mathematically sound framework. Network science is incredibly useful for anyone studying complex systems because their properties and structure are very often described and analyzed using network measures. Their applications to social sciences have been truly paradigm shifting, and they have potential to do the same for archaeology.

End-of-Chapter Exercises

1. We hard-coded in the radius of the network in our model `layout-circle sort turtles 10`. How could you change this to be dynamic, so that the radius automatically updates when you change the size of the window?

2. In many cases we do not want isolated nodes in our network. Use the properties of the `mean-path-length` primitive to develop an algorithm that adds a minimum number of edges necessary to ensure that the graph is complete.

3. Repeating the same code to calculate the number of links of four breeds is tiresome, especially if you decide to increase the number of lineages. Can you redesign the `GRN-setup` procedure?

4. Using a set of switches in the INTERFACE, examine how each of these exchange algorithms works. What is the distribution of pots with just pooling, with GRN, or with BRN? How do pooling, GRN, and BRN impact the average degree, betweenness, and shortest path measures?

5. Remember the profiler from chapter 1? The model developed above could be made more computationally efficient. Run the profiler and think about how can you improve its performance. For example, do you have to repeatedly query all turtles to then only have those with `pots > 1` perform actions?

Further Reading

Network science is its own discipline, so there are dozens of books and articles to recommend. Here are a few of our favorites.

- L. Borck et al. 2015. "Are Social Networks Survival Networks? An Example from the Late Pre-Hispanic US Southwest." *Journal of Archaeological Method and Theory* 22 (1): 33–57. doi:10.1007/s10816-014-9236-5

- T. Brughmans. 2010. "Connecting the Dots: Towards Archaeological Network Analysis." *Oxford Journal of Archaeology* 29, no. 3 (July): 277–303. doi:10.1111/j.1468-0092.2010.00349.x

- T. Brughmans, A. Collar, and F. Coward. 2016. *The Connected Past: Challenges to Network Studies in Archaeology and History*. Oxford, UK: Oxford University Press.

- T. Brughmans and M. Peeples. 2017. "Trends in Archaeological Network Research." *Journal of Historical Network Research* 1 (1): 1–24. https://jhnr.uni.lu/index.php/jhnr/article/view/10

- S. A. Crabtree. 2015. "Inferring Ancestral Pueblo Social Networks from Simulation in the Central Mesa Verde." *Journal of Archaeological Method and Theory* 22, no. 1 (March): 144–181. doi:10.1007/s10816-014-9233-8

- T. Evans. 2016. "Which Network Model Should I Use? Towards a Quantitative Comparison of Spatial Network Models in Archaeology." In *The Connected Past: Challenges to Network Studies in Archaeology and History,* edited by T. Brughmans, A. Collar, and F. Coward, 149–173. Oxford, UK: Oxford University Press.

- T. Froese, C. Gershenson, and L. R. Manzanilla. 2014. "Can Government Be Self-Organized? A Mathematical Model of the Collective Social Organization of Ancient Teotihuacan, Central Mexico." *PLOS ONE* 9 (1): e109966–e109966. doi:10.1371/journal.pone.0109966

- C. Knappett, T. Evans, and R. Rivers. 2011. "The Theran Eruption and Minoan Palatial Collapse: New Interpretations Gained from Modeling the Maritime Network." *Antiquity* 85 (329): 1008–1023. doi:10.1017/S0003598X00068459

- M. E. J. Newman. 2010. *Networks: An Introduction.* Oxford, UK: Oxford University Press.

- A. L. Barabási. 2016. *Network Science.* Cambridge, UK: Cambridge University Press.

- M. O. Jackson. 2008. *Social and Economic Networks.* Princeton, NJ: Princeton University Press.

For a truly thorough bibliography, see:
https://historicalnetworkresearch.org/bibliography/#Archaeology.

NOTES