3 Exploring and Extending Agent-Based Models

Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.

—Isaac Asimov

The perfect journey is never finished, the goal is always just across the next river, round the shoulder of the next mountain. There is always one more track to follow, one more mirage to explore.

—Rosita Forbes

Sailors on a becalmed sea, we sense the stirring of a breeze.

—Carl Sagan

In this chapter you will learn how to modify and extend an agent-based model. We will do this by taking several classic models, examining how they work, and then discussing how to change and extend these models. Sometimes the modified models explore alternative scenarios, variations on the original model. Other times, the modifications may lead to models of very different phenomena.

During these explorations it is important to pay attention to four characteristic features of agent-based modeling:

1. Simple rules can be used to generate complex phenomena

Many of the models that we will look into have very simple rules that do not require complex mathematical formulas or a deep understanding of the knowledge domain that they are attempting to model. Nonetheless, they are able to reproduce complex phenomena that are observed in the real world. For instance, a model of fire spread may have only a simple rule to describe fire spread from one tree to another, but it still may have interesting things to say about how likely a fire is to spread across an entire forest.

2. Randomness in individual behavior can result in consistent patterns of population behavior

It is common for people when they see an ordered population level behavior such as a flock of birds to assume that there must be deterministic processes that govern the behavior of the individuals (Wilensky & Resnick, 1999). In the case of the birds, people tend to believe that there must be specific social rules or communications that tell each bird how to place itself in the flock. However, nature has some surprises for us: Many times the individual level rules are quite simple (see point 1) and do not necessarily tell the bird where to position itself in the flock. Instead, the rules often contain a certain amount of nondeterminism and are robust to perturbations in the initial conditions. Despite the stochastic nature of these systems, they can still result in the generation of predictable high-level behavior like the flocking of birds.

3. Complex patterns can "self-organize" without any leader orchestrating the behavior

Similarly, it is common for people when they see a flock of birds, to assume that there must be a leader who orchestrates the behavior—a leader bird who tells each follower bird what to do (Resnick & Wilensky, 1993; Resnick, 1994a; Wilensky & Resnick, 1999).[1] However, nature again surprises: a population of individuals, each following very simple rules, can "self-organize," generating complex and beautiful patterns without any orchestrator or centralized controller—these patterns are called "*emergent*" (Wilensky & Reisman, 2006).

4. Different models emphasize different aspects of the world

Even after we have completed a good working model of a particular phenomenon, we have not finished with the modeling process. Every model foregrounds certain aspects of the world and backgrounds other aspects. There can be many models of the same phenomenon, and they may each have something interesting to say about the way the world works. For instance, a model of residential location preferences may emphasize how likely a person is to move into a neighborhood based on how she likes the other people in the neighborhood. Such a model may have very interesting things to say about how urban populations develop, but it may say nothing about school districts, location of retail businesses, or development of parks, all of which are also affected by residential location preferences.

In the rest of this chapter, we will dive right into several classical agent-based models and go through the steps of modifying and extending each of the models.

1. Together, points 2 and 3 make up what Resnick and Wilensky have called the "deterministic-centralized mindset." This mindset is a widespread tendency for people, when observing a population-level behavioral pattern, to explain it by postulating a centralized controller and nonrandom individual-level behavior (Wilensky & Resnick, 1999).

Figure 3.1
A forest fire consuming trees in a forest.

The Fire Model

Many complex systems tend to exhibit a phenomenon known as a "critical threshold" (Stauffer & Aharony, 1994) or a "tipping point" (Gladwell, 2000).[2] Essentially, a tipping point occurs when a small change in one parameter results in a large change in an outcome. One model that clearly contains a tipping point is the early agent-based model of a forest fire. This model is easy to understand, yet exhibits some interesting behavior. Besides being interesting in its own right, the model of forest fire spread is highly relevant to other natural phenomena such as the spread of a disease, percolation of oil in rock, or diffusion of information within a population (Newman, Girvan & Farmer, 2002).

This simple model is highly sensitive to one parameter. When observing the resultant outcome of whether or not a fire will burn from one side of a forest to another (percolate), the output is mainly dependent on the percentage of the ground that is covered by trees (see figure 3.1). As this parameter increases, there will be little to no effect on the system for a long time, but then all of a sudden the fire will leap across the world. This is a "tipping point" in the system. Knowing that a system has tipping points can be useful for a variety

2. There are several different terms from different fields that are used to describe tipping, for example, "phase transition" in physics (Stanley, 1971). In general, all of these terms refer to a small change of an input parameter resulting in a large change in some output variable of interest.

of reasons. First, if you know a system exhibits a tipping point, you know that continuing to put effort into the system, even if you are not seeing any results yet, may yet bear fruit. Second, if you know where the tipping point is, and if you know how close you are to it, then you can determine whether or not it is worth putting additional effort into the system. If you are far away from the tipping point, then it might not be worthwhile trying to change the state of the system, whereas if you are close to the tipping point it may take only a small amount of effort to make a big change in the state of the system.

Description of the Fire Model

The Fire model arose from a number of independent efforts to understand percolation phenomena. In percolation, a substance (such as oil) moves through another material (such as rock), which has some porosity. Broadbent and Hammersley (1957) first posed this problem, and since then many mathematicians and physicists have worked on it. Influenced by cellular automata models, they introduced a percolation model using the example of a porous stone immersed in a bucket of water. The question they focused on was: What is the probability that the center of the stone becomes wet?

A fire moving through a forest can be thought of as a kind of percolation where the fire is like the oil and the forest is like the rock, with the empty places in the forest analogous to the porosity of the rock. A similar question to Broadbent and Hammersley's is: If you start with some burning trees on one edge of the forest, how likely is the fire to spread all the way to the other side of the forest? Many scientists created and studied such fire models. In 1987, the Danish physicist and complex systems theorist Per Bak and his colleagues showed that the spread of the fire depended on a critical parameter, the density of the forest. Because this parameter arises naturally, the complexity of the fire can arise spontaneously and was therefore a possible mechanism to explain the world's naturally arising complexity. Bak and his colleagues called this phenomenon "self-organizing-criticality" and demonstrated it in a number of contexts including, famously, the emergence of avalanches in sand piles.

Herein, we will explore a version of the Fire model adapted by Wilensky (1997a) that is developed in the NetLogo modeling language and is distributed in the Earth Science section of the NetLogo models library. You can find the simplified version of the Fire model in Sample Models > IABM Textbook > Chapter 3 > Fire Extensions > Fire Simple .nlogo. This version of the Fire model contains only patches, no turtles. The patches can have four distinct states. They can be (1) green, indicating an unburned tree, (2) red, indicating a burning tree, (3) brown, indicating a burned tree, or (4) black, indicating empty space (illustrated in figure 3.2). When the model is first set up the left edge of the world is all red, indicating that it is "on fire." When the model starts running, the fire will ignite any "neighboring" tree—that is, a tree to its right, left, below, or above it that is not already burned and not already on fire. This will continue until the fire runs out of trees that it can ignite. The only "control parameter" in this model is the density of trees in the world. This

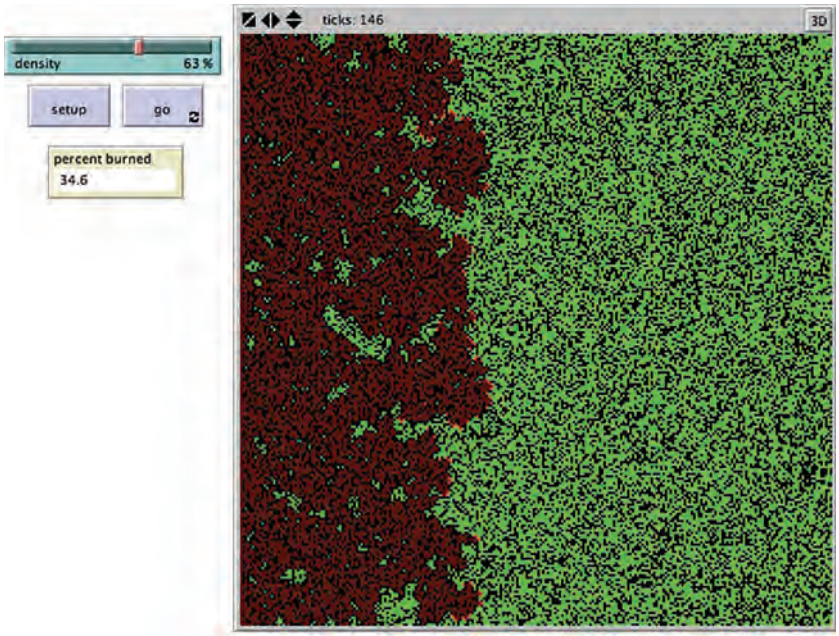

Figure 3.2
NetLogo Fire Simple Model. Based on NetLogo Fire model (Wilensky, 1997). http://ccl.northwestern.edu/netlogo/models/Fire.

density parameter is not an exact measure of the number of trees in the world. Rather, it is a probability that determines whether or not each patch in the world contains a tree. Because the density is probabilistic and not deterministic, even if you run the model multiple times with the same density setting, you will get different results.

Let us examine the simple rules that govern the Fire model. The code that initializes the model is as follows:

```
to setup
    clear-all
       ;; make some green trees
    ask patches [
       if (random 100) < density
           [ set pcolor green ]
       ;; make a column of burning trees at the left-edge
       if pxcor = min-pxcor
           [ set pcolor red ]
    ]
    ;; keep track of how many trees there are
    set initial-trees count patches with [pcolor = green]
    reset-ticks
end
```

Box 3.1
Modeling Choices

> This way of setting up the trees is only one of many possible ways to do it. Can you write rules so that the same number of trees will be created on each run of the model? Can you think of other ways of initializing the trees?

Just as in the Life Simple model we saw in chapter 2, the Fire Simple model will use only stationary agents, patches, and no moving agents, turtles. Besides these basic types of agents, agent-based models can have many other types, including user created agent types. We will discuss these different types of agents in detail in chapter 5.

The first line of the procedure is a command, CLEAR-ALL (or CA for short in the NetLogo language), which resets the world to its initial state—it resets the model's clock, kills all moving agents, and restores the default values to the stationary agents. The rest of this code issues commands to the patches. First, it populates the world with trees, and second, it makes a column of burning trees (by setting their color to red, the indication that they are on fire) at the left edge of the world. We have made many modeling choices here. Foremost among these are (1) modeling empty space in the forest as black patches, (2) modeling nonburning trees in the forest as green patches, and (3) modeling fire as burning trees, which are represented as red patches.

Once these modeling choices are made, we can write the code to populate the model. To set up the trees, we have asked the patches to perform:

```
if (random 100) < density
    [ set pcolor green ]
```

To help us in understanding the setup code, suppose that the density is set to 50. This code tells each patch to roll a hundred-sided "die" (or, an alternative metaphor, spin a spinner with 100 equal sectors). Let us take the point of view of a patch. If I'm a patch, I throw the die. If that die comes up less than 50, then "I become a tree," otherwise "I don't do anything." So, half of the time I will become a tree (turn green) and the other half of the time I'll remain black. Note that each patch throws its own independent die so theoretically all of them (or none of them) could become trees. But, on average, half will become trees. If the density were higher or lower, the same code would work to populate the model with roughly the appropriate tree density. This trick of taking the point of view of the agent, what we called agent-centric thinking in the last chapter, is one we will employ frequently and is a good habit to form for facility with ABM.

The fact that the behavior of an agent-based model can vary from run to run is an important aspect of agent-based modeling. It means that one run is never enough to truly capture the behavior of most ABMs—instead, you have to perform multiple runs and aggregate the results. How to perform this aggregation will be discussed in greater depth in future chapters.

Once the model is set up or initialized, we must define what the model does at each "tick" of the clock. This is typically done in a procedure named GO.

The core of the GO procedure for the Fire Simple model is:

```
to go
    ;; ask the burning trees to set fire to any
    ;; neighboring (in the 4 cardinal directions) non-burning trees
    ;; the "with" primitive restricts the set of agents to
    ;; those that satisfy the predicate in the brackets
    ask patches with [ pcolor = red ] [
       ask neighbors4 with [ pcolor = green ] [
           set pcolor red
       ]
    ]
    ;; advance the clock by one "tick"
    tick
end
```

The code is short. It asks all the fire agents (which are the red patches) to ignite their neighboring unburned tree (green) patches by setting their color to red. (The neighbors4 primitive specifies those neighbors in the cardinal directions, i.e., north, south, east and west). Finally, it tells the clock to advance one tick.

Running this GO procedure repeatedly (as described earlier) will produce a working fire model. For visualization purposes it is useful to distinguish trees that have just caught fire from trees that are all burned up. We therefore add one more line to the model:

```
set pcolor red - 3.5
```

which, as we saw with the Life Simple model in chapter 2, darkens the color of the burned trees.

As currently written the GO procedure will never stop running. To fix that, we add a stopping condition so that the model stops when all of the fires are burned out. This is added to the beginning of the GO procedure:

```
if all? patches [pcolor != red]
      [ stop ]
```

The code for the GO procedure is then as follows:

```
to go
    ;; stop the model when done
    if all? patches [pcolor != red]
       [ stop ]

    ;; ask the burning trees to set fire to any neighboring non-burning trees
    ask patches with [pcolor = red] [ ;; ask the burning trees
       ask neighbors4 with [pcolor = green] [ ;; ask their non-burning neighbor trees
       set pcolor red ]   ;; to catch on fire
    set pcolor red - 3.5    ;; once the tree is burned, darken its color
]
tick  ;; advance the clock by one "tick"
end
```

This model is now functionally complete. The Fire Simple model included in the IABM Textbook folder in the models library also sets up some variables so as to display the percentage of the trees that have been burned.

In the following box, we describe the two basic types of primitives and procedures in NetLogo, commands and reporters. It is often handy to write reporter procedures to shorten code. For example, we might want to shorten the code calculating the patches that are on fire. We can create a reporter, called e.g., fire-patches, that calculates these. To do, we use the special NetLogo primitives, "report" and "to-report."

```
to-report fire-patches
    report patches with [pcolor = red]
end
```

Now we can change the code from the GO procedure above:

```
ask patches with [pcolor = red] [
```

to

```
ask fire-patches [
```

In this case, creating the reporter only slightly reduced the length of the code, but in more complex cases, encapsulating code as reporters can both shorten and clarify the code. There are many other good reasons to use reporters.

Box 3.2
Commands and Reporters

In NetLogo, *commands* and *reporters* tell agents what to do. A command is an action for an agent to carry out, resulting in some effect. A reporter is instructions for computing a value, which the agent then "reports" to whoever asked it.

Typically, a command name begins with a verb, such as "create," "die," "jump," "inspect," or "clear." Most reporter names are nouns or noun phrases.

For example, "forward" is a command, it tells a turtle agent to carry out a movement. But "heading" is a reporter, it tells the agent to report the angle it is facing.

Many commands and reporters are built into NetLogo, and you can create new ones in the Code tab. Commands and reporters built into NetLogo are called primitives. The NetLogo Dictionary has a complete list of built-in commands and reporters.

You can create your own commands and reporters by defining them in procedures. Each procedure has a name, preceded by the keyword to or to-report, depending on whether it is a command procedure or a reporter procedure. The keyword end marks the end of the procedure. It can be useful to notice that reporters are colored purple and commands are colored blue in NetLogo's automatic syntax highlighting.

Sometimes an input to a primitive is a *command block* (zero or more commands inside square brackets, e.g., [forward 10 set color red]) or a *reporter block* (a single reporter expression inside square brackets, e.g., the Boolean reporter [color = red]).

For more information on commands and reporters, see http://ccl.northwestern.edu/netlogo/docs/programming.html#procedures/.

- By creating a reporter that we can reuse, we avoid repetition of the code and therefore minimize the risks of inconsistency.
- The name of the reporter procedure can serve as self-documentation; a good procedure name clarifies intent.
- Isolate complexity: allows reasoning about, e.g., a GO procedure without being burdened by details of what goes on inside the reporter code.

We examine the use of reporters a little further in the explorations for this chapter.

Now run the Fire Simple model several times. If we run it with a low density of trees, we will see, as expected, very little spread of the fire. If we run it with a very high density of trees, we will see, as expected, the forest being decimated by the inexorable march of the fire. What should we expect at medium densities? Many people surmise that if the density is set to 50 percent, then the fire will have a 50 percent probability of reaching the right edge of the forest. If we try it, however, we see that at 50 percent density, the fire does not spread much. If we raise it to 57 percent, the fire burns more, but still doesn't usually reach the other side of the forest (see figure 3.3). However, if we raise the density to 61 percent, just 2 percent more density, the fire inevitably reaches the other side (see figure 3.4). This is unexpected. We would expect a small change in density to have a

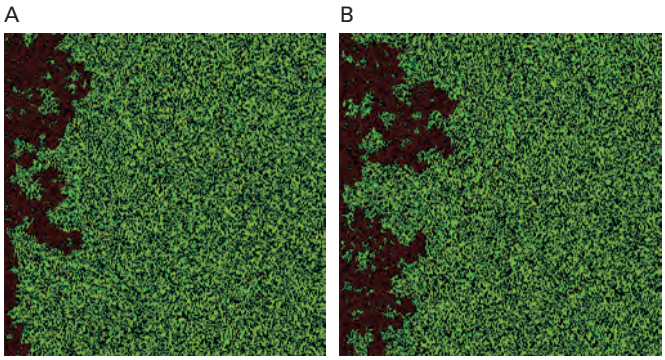

Figure 3.3
Two typical runs of the Fire Simple model with density set to 57 percent.

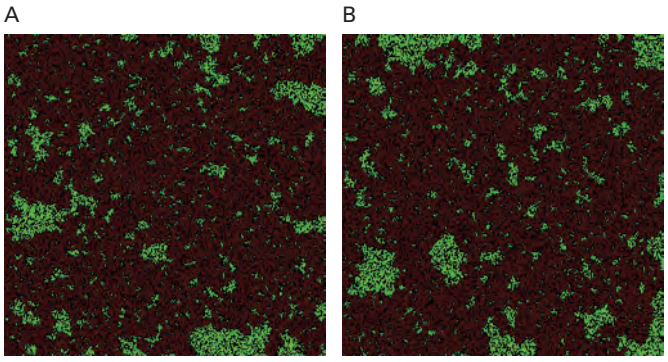

Figure 3.4
Two typical runs of the Fire Simple model with density set to 61 percent.

relatively small effect on the spread of the fire. But, it turns out, the Fire model has a "critical parameter" of 59 percent density. Below 59 percent density the fire does not spread that much; above it, it spreads dramatically farther. This is an important and prevalent property of complex systems: They exhibit nonlinear behavior where a small change in input can lead to a very large change in output.

In the next three sections, we will be creating variants/extensions to the Fire Simple model. These extensions can be found in the NetLogo models library, in the IABM folder, in the subfolder Chapter Three > Fire Extensions.

First Extension: Probabilistic Transitions

The Fire Simple model, like any model, makes many simplifying assumptions about the way the world works. In particular, fire in the real world does not spread deterministically

from one tree to another tree, but rather it jumps to another tree based on a variety of factors such as wind, the type of wood, even how close the branches are to each other. Often when there are many factors that affect a process like this one, we simplify the mechanism by using random numbers. We know that it is not certain that the fire will spread from one tree to another, but we do not know exactly what the probability of the spread is. We can model the mechanism by which fire spreads from one tree to another via a probability, and we can make this probability a parameter of the system. By experimenting with different values of the probability we can explore what effect the probability has on the overall spread of the fire.

To do this we first create a slider in the NetLogo model called PROBABILITY-OF-SPREAD. We set up this slider to go from 0 to 100. This is the variable that will control how frequently the fire spreads from one tree to another. Simply adding the slider is not enough—we need to modify the code itself to take this parameter into account. To do that we must first decide how we want this parameter to affect the model. Every time the model "spreads" the fire, we want it to create a random number between 0 and 100, if that number is less than the PROBABILITY-OF-SPREAD then we spread the fire as before. If it is not, then the fire does not spread in that direction. Let's examine the GO procedure in the original model. The fire spread occurs when all the fire agents (which are the red patches) ignite their neighboring tree patches by setting their color to red. This seems like the logical place to insert our new variable. The old code looks like this:

```
ask patches with [pcolor = red] [
    ask neighbors4 with [pcolor = green] [
         set pcolor red
    ]
    …
```

To modify this model to take our new parameter into account, all we have to do is add one additional check:

```
ask patches with [pcolor = red] [
                 ask neighbors4 with [pcolor = green] [

        ;; only burn if a random draw is greater than the probability of spread
                          if random 100 < probability-of-spread [ set pcolor red ]
                 ]
…
```

As before the fire agents look to their four neighbors and pick out those that are unburned trees. But this time, instead of just burning those trees, each tree executes RANDOM 100 which will generate an integer between 0 and 99; if that number is less than the value of PROBABILITY-OF-SPREAD the model proceeds as before. If not, then

the fire does not spread to that tree. It is important to note that even if one of these probabilities "fails" it is still possible that the tree in question may be burned down by one of its other burning neighbors.

Try setting both the DENSITY and the PROBABILITY-OF-SPREAD parameters to a variety of values. You can reproduce the results of the original model by setting the PROBABILITY-OF-SPREAD slider to 100 percent. It is a good rule of thumb to extend your model so you can reproduce previous results. By doing so, you can check to see if your new model is consistent with your old model and verify that you did not introduce any coding errors into the new model. You will often want to be able to generate the old results in order to compare them to the new results—ensuring that the new model can replicate the old mechanisms means you do not have to keep multiple versions of the model around.

Change these parameters and let the model run a few times. What happens to the model? If you set the tree densities to values that, in the original model, would have allowed the fire to burn through the forest, and set the PROBABILITY-OF-SPREAD at 50 percent, the fire rarely burns through the forest. In fact, with a 50 percent spread probability, it takes a very high density for the fire to burn completely through the forest. While experimenting, you might now notice that the density parameter has an upper bound of 99 percent. In the original model, this made sense as 100 percent density was uninteresting, but now, since fire spread between trees is probabilistic, 100 percent density is also interesting. Modify the upper bound of the slider, set the density to 100 percent and observe the fire spread. Our new parameter has dramatically altered the numerical results of the model. A screenshot of the Fire Simple model after completing this first extension is shown in figure 3.5.

Second Extension: Adding Wind

Sometimes we can take an ABM for which we used a probability to model a suite of mechanisms and refine it so that some of those mechanisms are modeled in a more physical way. Our first extension hides a host of possible mechanisms in the PROBABILITY-OF-SPREAD. We can pull out one of those mechanisms and model it in a more refined way. Wind is a good example of a process that we can model more specifically. We can think of the effect of wind on a fire as increasing the chance of fire spread in the direction it is blowing, decreasing the chance of fire spread in the direction it is not blowing, and having little effect on fire spread that occurs perpendicular to the direction of its movement. Of course, this too is an oversimplification—there are often local effects of wind, such as turbulence. As we have seen, all models are simplifications, and playing the modeling game means we will accept this one for now.

To implement wind in our model, we will create two sliders. One will control the speed of the wind from the south (a negative value will indicate a wind from the north)

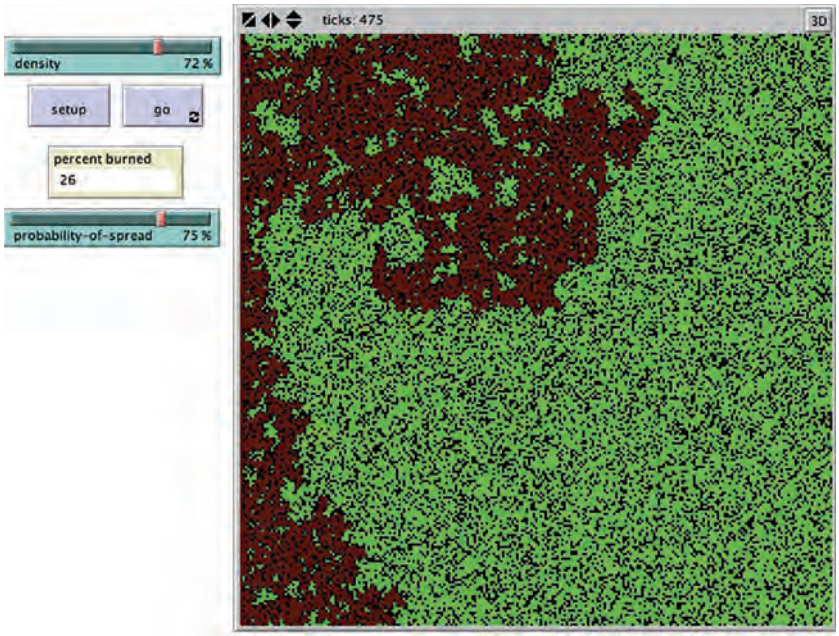

Figure 3.5
NetLogo Fire Simple model after first extension.

and one will control the speed of the wind from the west (a negative value will indicate a wind from the east). We create these two sliders and set them up to go from –25 to 25. How do we use these new parameters in the code? If we think back to our first modification, we want these new parameters to affect the PROBABILITY-OF-SPREAD. This effect will be based on the direction that the fire is attempting to spread. Since we have set the wind speed to vary from –25 to 25, we could conceive of those numbers as percentages by which to modify the probability of spread which is expressed as a percent. In order to do this, let us first create a local variable called PROBABILITY that will initially be set to PROBABILITY-OF-SPREAD. A local variable is one that has a value only within a limited context, usually inside the procedure in which it is defined. If we only need to reference a variable in one procedure, then it is best to use a local variable. But, remember, you will not be able to see the value of the variable elsewhere in your program or in the command center. We define a local variable with the "let" primitive and can modify it with the "set" primitive. We can modify PROBABILITY to take into account WIND-SPEED by increasing or decreasing it by the WIND-SPEED in the direction the fire is burning. When we put this all together, we get the following code:

```
to go

    if all? patches [pcolor = red]
         [ stop ]

    ;; each burning tree (red patch) checks its 4 neighbors.
    ;; If any are unburned trees (green patches), change their probability
    ;; of igniting based on the wind direction
    ask patches with [pcolor = red] [
         ;; ask the unburned trees neighboring the burning tree
         ask neighbors4 with [pcolor = green] [
             let probability probability-of-spread     ;; define a local variable

             ;; compute the direction you (the green tree) are from the burning tree
             ;; (NOTE: "myself" is the burning tree (the red patch) that asked you
             ;; to execute commands)
             let direction towards myself
             ;; the burning tree is north of you
             ;; so the south wind impedes the fire spreading to you
             ;; so reduce the probability of spread
             if (direction = 0 ) [
                  set probability probability - south-wind-speed ]

             ;; the burning tree is east of you
             ;; so the west wind impedes the fire spreading to you
             ;; so reduce the probability of spread
             if (direction = 90 ) [
                  set probability probability - west-wind-speed ]

             ;; the burning tree is south of you
             ;; so the south wind aids the fire spreading to you
             ;; so increase the probability of spread
             if (direction = 180 ) [
                  set probability probability + south-wind-speed ]

             ;; the burning tree is west of you
             ;; so the west wind aids the fire spreading to you
             ;; so increase the probability of spread
             if (direction = 270 ) [
                  set probability probability + west-wind-speed ]
             if random 100 < probability [ set pcolor red ]
         ]
         set pcolor red - 3.5
    ]
    tick
end
```

This code is a little tricky. Essentially, what it does is modify the probability of spread, increasing it in the direction of the wind and decreasing it in the opposite direction. It calculates the change in the probability by first determining which direction the fire is trying to spread and then determining which of the winds will affect it. Once this probability is calculated, it is then used to determine whether or not the fire spreads to the neighboring tree.

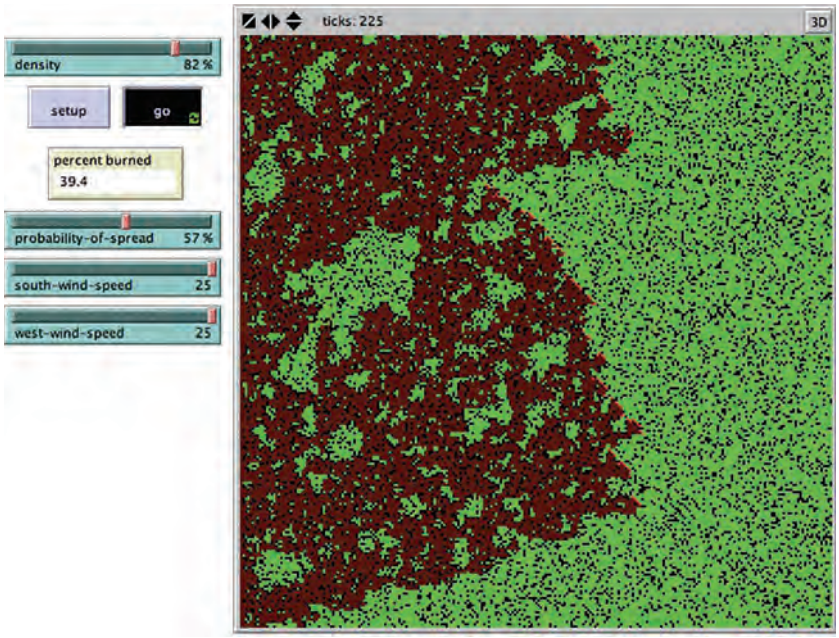

Figure 3.6
NetLogo Fire model after second extension.

This modification can lead to quite interesting patterns of spread. For example, set the density at 100 percent, and make the wind blow strong from the south and west. At the same time, set the PROBABILITY-OF-SPREAD fairly low, say around 38 percent. This creates a triangular spread, and, all else being equal, the fire should spread to the northeast. (See figure 3.6.)

Third Extension: Allow Long-Distance Transmission

We have modeled the effect of wind as pushing the fire to spread in one direction. Another possible effect of wind is to enable the fire to jump over long distances and start fires where there are no surrounding burning trees. That might be an interesting process to model. In order to make sure the revised model can replicate our old results, we add a switch to control the jumping. We add a Boolean (TRUE or FALSE) switch labeled BIG-JUMPS? Switches in NetLogo provide a way to have direct control over variables that can only be true or false. This particular switch allows us to turn on and off the jumping behavior. With the switch off, the revised model should produce the same results as Fire Simple Extension 2. One way to model fire jumping due to wind would be to ignite a new fire at some distance in the direction of the wind. We can do this by modifying the code inside our earlier GO procedure as follows picking up after we do all of the changes in probability based on wind:

```
…
if random 100 < probability [
                    set pcolor red

                    ;; if big-jumps is on, then sparks can fly farther
                    if big-jumps? [
                             let target patch-at ( west-wind-speed / 5 ) ( south-
                             wind-speed / 5 )
                             if target != nobody and [pcolor] of target = green [
                             ask target [
                                 set pcolor red ;; ignite the target patch
                             ]
                             ]
                    ]
            ]
…
```

The first part of this code turns the current patch to red, which is the same as the previous version, but the second part after IF BIG-JUMPS? is what has changed. If BIG-JUMPS? is true then it looks for a target patch that is some distance away in the direction of the wind. If there is an unburned (green) tree at that location, then that tree is also set on fire (due to the spark having landed there). A more detailed model could include explicit spark agents, which travel according to the wind, and catch trees on fire when they land, but in this case we just model the effects of flying sparks without modeling the sparks themselves.

Insert this code and see what effect it has on the model. You start to see lines in the direction of the wind as it jumps across gaps in the forest (as shown in figure 3.7). This extension can have visually dramatic effects. Explore different sets of parameters and observe the different patterns of spread you can get. This modification increases the probability that the fire will reach the right edge of the world (our measure), but the resultant pattern is no longer the same; there are big chunks of the world that are not burned anymore. As new patterns emerge we may need to reevaluate the questions that drove us to create the model, which may change the measures we collect about the model.

Each extension can lead to many more questions, which in turn call for new extensions. In our third fire extension, it is of course also possible to have big jumps be probabilistic in the same way that the neighboring spreads are, or to have the sparks from the big jumps land in a random rather than a determined location. Both of these modifications are left as exercises to the reader.

Summary of the Fire Model

We have made three different changes to the model, which have affected our tipping point in three different ways. The final change to the model points out that the measure we were observing (whether or not the fire made it to the right edge of the world) might not even be the measure we want to observe. As we change the fire spread mechanisms, other measures such as “percent burned” may become more important. These new measures

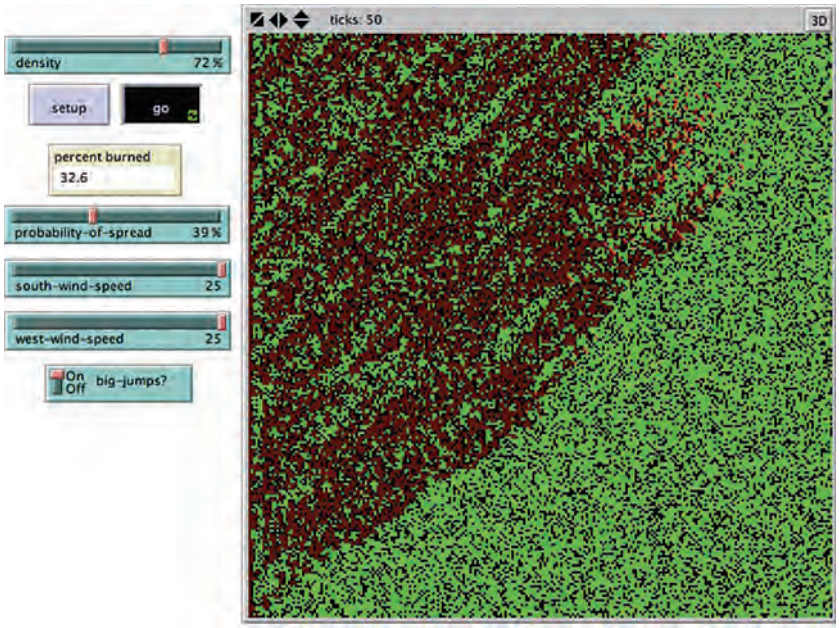

Figure 3.7
NetLogo Fire model after third extension.

may or may not have tipping points. This illustrates an important feature of tipping points. They are defined by an input and an output measure. A model does not have a tipping point in and of itself—the tipping point is relative to the choice of inputs and output measures.

Advanced Modeling Applications

As we mentioned, the Fire model can be generalized into a model of percolation, which is a well-studied problem in geology and physics (Grimmett, 1999). The generalized percolation form is: Given a structure with probabilistic transitions between locations, how likely is it that an entity entering that structure will make it from one side of the structure to the other? (See figure 3.8.) This form of the question has obvious ramifications in the area of oil drilling (Sahimi, 1994). Yet these generalized percolation forms can be useful well outside of traditional percolation contexts, such as studying the diffusion of innovation. Percolation bears a resemblance to innovation diffusion (Mort, 1991) and can be used to model it. Many things can spread in communities. Communities can often spread diseases and, in this way, percolation can be relevant to epidemiology (Moore & Newman, 2000). Newman, Girvan, & Farmer (2002) have studied so-called highly optimized tolerant system such as forests that are robust in the face of fires. They use the degree of

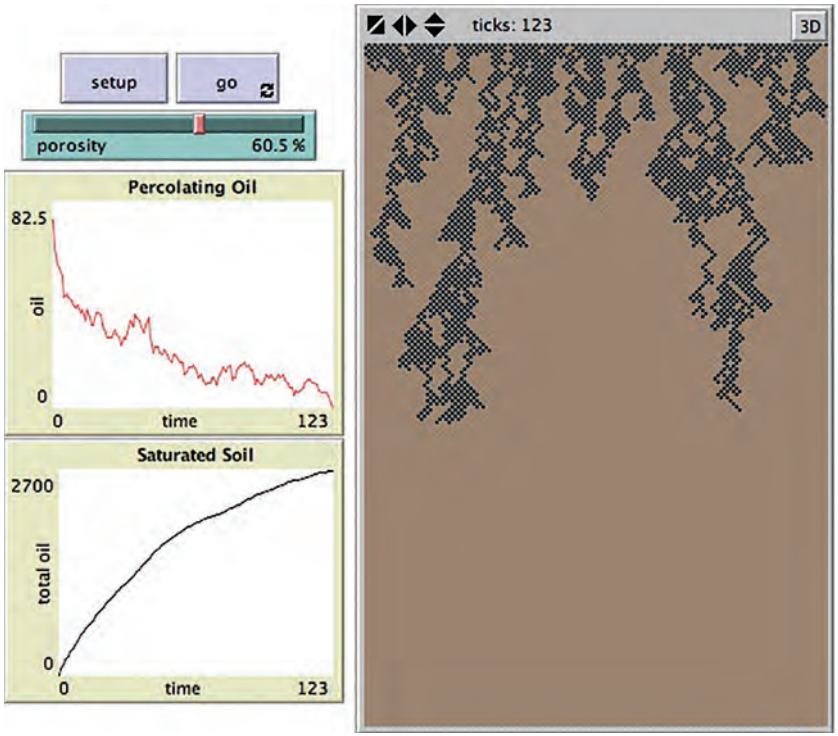

Figure 3.8
NetLogo Percolation model. http://ccl.northwestern.edu/netlogo/models/Percolation (*Wilensky 1998*).

percolation as a measure of the robustness of the system. These are but a few examples of the percolation of percolation models into other domains. Starting with the simple Fire model that we have examined, it is possible to model and gain insight into a wide variety of different phenomena.

The Diffusion-Limited Aggregation (DLA) Model

As we discussed in chapter 0, the ABM perspective enables a new and different understanding of complex systems. As demonstrated with the Fire Simple model, it is possible to think about inanimate objects (such as trees) as agents. By reconceptualizing atoms and molecules as agents, and then developing procedural rules to describe how these agents interact we can gain a deeper understanding of many different types of physical phenomena.

For instance, the formation of complex beautiful patterns in nature has mystified and amazed humans for many years (see figure 3.9), yet many of these patterns can be

Figure 3.9
A DLA copper aggregate formed from a copper sulfate solution in an electrode position cell (Kevin R. Johnson, 2006, http://commons.wikimedia.org/wiki/File:DLA_Cluster.JPG).

generated using simple rules. Often in ABMs we will see that the complexity of a resulting pattern bears little direct relationship to the complexity of the underlying rules. In fact, in some cases the results will seem to be exactly the opposite; simpler rules will often create more complex patterns. The focus of our interest should not necessarily be on the complexity of the rules, but instead on the interaction those rules produce. Many agent-based models are interesting not because of what each agent does, but because of what the agents do together.

In this section, we will look at a very simple model where all that the agents do is move randomly around the world, and eventually stop moving when a basic condition is met; despite the simplicity of the rules, interesting complex phenomena can emerge.

Description of Diffusion-Limited Aggregation

In many physical processes from the creation of clouds, to snowflakes, to soot, smoke, and dust, particles aggregate in interesting ways. Diffusion-limited aggregation (DLA) is an idealization of this process, and was first examined as a computational model in the early eighties (Witten & Sander, 1981, 1983). DLA models were able to generate patterns that resemble many found in nature, such as crystals, coral, fungi, lightning, and growth of human lungs, as well as social patterns such as growth of cities (Garcia-Ruiz et al., 1993; Bentley & Humphreys, 1962; Batty & Longley, 1994).

A NetLogo version of diffusion-limited aggregation was one of the first models written for the NetLogo models library (*Wilensky, 1997b*). You can find a simplified DLA model in the NetLogo models library under Sample Models > IABM Textbook > Chapter Three > DLA Extensions > DLA Simple.nlogo. This model starts with a large number of red particle agents and one green patch in the center. The green patch at the center of the screen is stationary, but when you press GO, all of the red particles move randomly by "wiggling" left and right (resulting in a small random turn) and then moving forward one step. After a particle moves, if any of its neighbors (i.e., nearby patches) are green, the particle dies, turning the patch it is on green (the particle can do so because all turtles have direct access to the variables of the patch they are on). If you let the model run long enough, you will get an interesting fractal-like pattern forming from the green patches (see figure 3.10).

Here is the NetLogo code for the GO procedure for DLA Simple. WIGGLE-ANGLE is a global variable, the value of which is set by a slider in the interface.

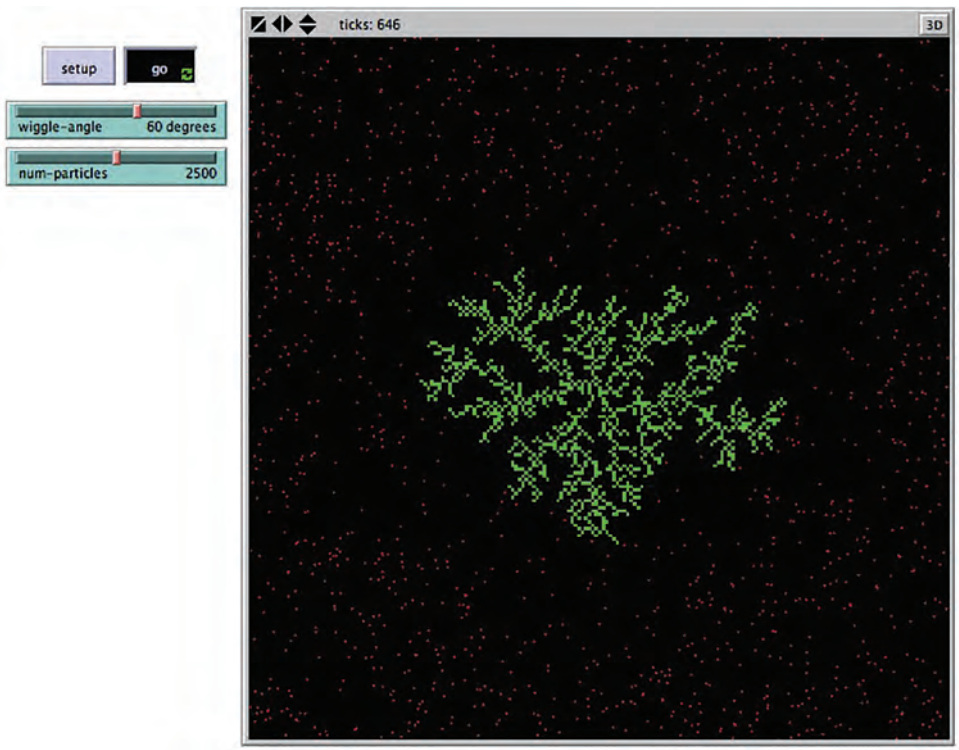

Figure 3.10
NetLogo model of diffusion limited aggregation. http://ccl.northwestern.edu/netlogo/models/DLA (based on *Wilensky, 1997b*).

```
to go
    ask turtles
       ;; turn a random amount right and left
       [ right random wiggle-angle
           left random wiggle-angle
           forward 1
           ;; if you are touching a green patch, turn your own patch green and
           ;; then die
           if ( any? neighbors with [pcolor = green] )
              [ set pcolor green
                  die ] ]
    tick
end
```

As the code shows, if any of the neighbors are green the particle stops moving, changes the color of the patch that it is on to green and dies. Turtles have direct access to the variables of the patch they are on—that is how the turtle can set the color of the patch it is on (the PCOLOR in NetLogo).

We will now create three extensions of the DLA Simple model. They are available in the folder Sample Models > IABM Textbook > Chapter Three > DLA Extensions.

First Extension: Probabilistic Sticking

The DLA Simple model has very simple rules. The code consists of only two procedures, and both of them are very small, yet these two procedures can generate many interesting results. In these extensions we will examine how adding a few more simple rules can enable us to generate more interesting patterns.

After you run the model for a while, you will realize that it always produces thin and wispy types of structures. Often the "stems" and "trunks" of the structures are only a single patch wide. This is because as soon as a particle touches anything green it will stop moving. It is much more likely that it will touch something toward the edge of the structure rather than near the interior of the structure.

However, we can change this. One way to think of this system is that if a particle comes into contact with a stationary object, then it becomes stationary itself with 100 percent probability. What if we decrease this probability? That is exactly what we will do in this extension. We will allow the user to control the probability of a particle becoming stationary.

In the original code, if any of the neighbors are green, the particle stops moving, changes the color of the patch that it is on to green, and dies. What we need to do is add another test to this rule, but we also have to make one other change to the code at the same time. Since stopping will be probabilistic, it is now possible that a particle could be on top of another green particle, and, in that case, we do not want the particle to stop. So we also need to make sure we are not on a green particle before we stop, which involves adding an additional condition to the model:

```
to go
    ask turtles
       [ right random wiggle-angle
           left random wiggle-angle
           forward 1
           if (pcolor = black) and ( any? neighbors with [pcolor = green] )
    and ( random-float 1.0 < probability-of-sticking )
              [ set pcolor green
                                    die ] ]
    tick
end
```

Now, a particle will only stick if a random number it generates is less than a given probability, but where is this probability defined? If you add this code into the GO procedure and then move to the Interface tab, NetLogo will generate an error because we have not defined this parameter yet. So we will do that now. We go back to the Interface tab and create a slider called PROBABILITY-OF-STICKING, and we give it a minimum value of 0.0, a maximum value of 1.0 and an increment of 0.01.[3] Now we are finished with this extension, and we can run the model with a PROBABILITY-OF-STICKING of 0.5, as seen in figure 3.11.

You will notice that the branches of the structure become thicker when the probability is 0.5. This is because particles have a smaller probability of getting stuck on the outside of the structure and can wander deeper in to the structure before stopping. If you want you can still generate the original results by setting the probability to 1.0. As with the extensions to the Fire Simple model, it is often helpful to make sure that when extending a model you can still generate the previous results. Not only does this allow you to inspect whether any errors were introduced by your code change, but also, as you continue to add new parameters and mechanisms, you will often want to go back to your original results for comparison.

Second Extension: Neighbor Influence

The addition of our probability parameter enables us to explore a whole new range of structures, while not interfering with our ability to add additional simple rules to this model. One commonly explored extension to the DLA model is to explore how the probability of sticking is related to the number of neighbors that are already stationary (the green patches) (Witten & Sander, 1983). If a particle is moving, it is more likely that it will stick somewhere if two or three of its neighbors are at rest than if just one of its neighbors is at rest. Thus, we want the probability of stopping to increase as the number of stopped neighbors increases.

Let us begin this extension by adding a switch to our interface, which we will call NEIGHBOR-INFLUENCE? This switch will determine whether or not we are taking the

3. Whereas the "random" primitive generates random whole number, here we use the "random-float" primitive so we can get probability values between 0 and 1.

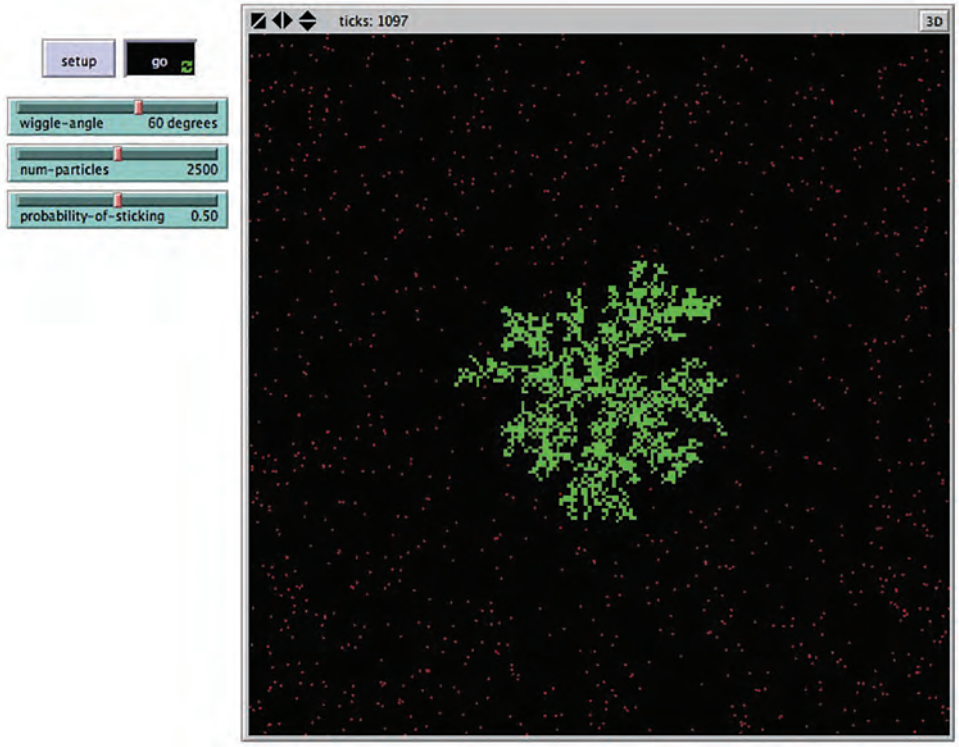

Figure 3.11
DLA model after first extension.

number of neighbors in to account when determining if a particle should stop moving. NEIGHBOR-INFLUENCE? is a Boolean variable, that is, it has one of two values, TRUE or FALSE. It is conventional to append a question mark character to the end of the variable name to point out to the reader that the variable is Boolean. After we add this switch we can go back and look at our GO procedure. We left the GO procedure looking like this:

```
to go
    ask turtles
       [ right random wiggle-angle
           left random wiggle-angle
           forward 1
           if ( pcolor = black ) and ( any? neighbors with [pcolor = green] ) and
                 ( random-float 1.0 < probability-of-sticking )
                 [ set pcolor green
                 die ] ]
    tick
end
```

We need to modify the PROBABILITY-OF-STICKING based on the NEIGHBOR-INFLUENCE? switch. To accomplish this we must first create a variable, which we will call LOCAL-PROB If NEIGHBOR-INFLUENCE? is turned off then LOCAL-PROB will be the same as PROBABILITY-OF-STICKING. However, if NEIGHBOR-INFLUENCE? is turned on, then we will reduce the probability of sticking for small numbers of green neighbors. We accomplish this by multiplying the PROBABILITY-OF-STICKING by the fraction of its eight neighbors that are green. For example, if PROBABILITY-OF-STICKING is set to 0.5 and if a particle encounters a neighborhood with four green neighbors, then we multiply PROBABILITY-OF-STICKING by 4/8 to get a sticking probability for that encounter of 0.25. We do this in order to create a relationship such that if many neighbors are green the probability approaches the PROBABILITY-OF-STICKING and if few neighbors are green the probability approaches zero.[4] As a further exploration, try some other functions and see how it affects this result; to facilitate this, you may want to make the number of neighbors that affect a particle a parameter. The new code looks like this:

```
      to go
            ask turtles
[ right random wiggle-angle
    left random wiggle-angle
    forward 1
    let local-prob probability-of-sticking

    ;; if neighbor-influence is TRUE then make the probability proportionate
    ;; to the number of green neighbors, otherwise use the slider as before
    if neighbor-influence? [
    ;; increase the probability of sticking the more green neighbors there are
                        set local-prob probability-of-sticking *
(count neighbors with [pcolor = green] / 8)
    ]

    if (pcolor = black) and ( any? neighbors with [pcolor = green] )
and ( random-float 1.0 < local-prob )
    [ set pcolor green
          die ] ]
    tick
end
```

When we run the model with this change in the code, we notice (1) that it takes much longer to form a structure because there is a much lower probability of a mobile particle stopping, and (2) the structures that emerge are very thick and almost bloblike. In addition there are many fewer one patch-wide branches, because the probability of stopping when in contact with only one other stationary patch is very small. You can see the results in figure 3.12.

4. Note that, in doing so, we limit the probability of sticking with only one green neighbor to one-eighth of the probability set in the slider.

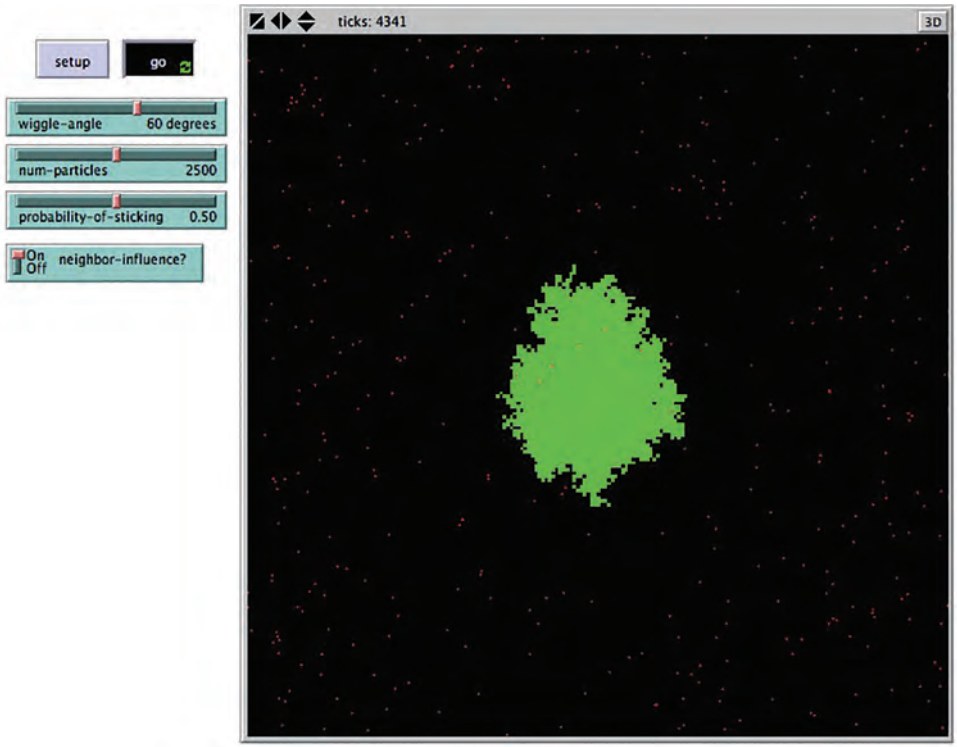

Figure 3.12
DLA model after second extension.

Third Extension: Different Aggregates

The previous two extensions have concentrated on deciding how a particle should stop when it is moving—there is another way that we can control how a particle stops. We can simply give it more places to stop at the beginning. If we look at the SETUP procedure in the previous version of the model, we can see that it creates one green patch in the middle of the world:

```
to setup
    clear-all
    ;; ask the middle patch to turn green
    ask patch 0 0
       [ set pcolor green ]
    create-turtles num-particles
       [ set color red
           set size 1.5   ;; make the particle bigger so it's easier to see
           setxy random-xcor random-ycor ]
end
```

Patch 0 0 is the unique name for the patch that is at x-coordinate 0, and y-coordinate 0, which is at the center of the NetLogo world by default. However, what if we want to create multiple green patches at the start? If there are multiple green patches, then there will be more places for the moving particles to come to rest, and we can generate different patterns of aggregation. To begin with, we need to create a slider so we can control the number of different aggregates that we create; we will call this slider NUM-SEEDS. We give this slider a minimum of 1 (since we need at least one seed), a maximum of 10, and an increment of 1.

Once we have this slider, we can ask NUM-SEEDS patches to turn green in the setup:

```
to setup
    clear-all
    ;; start with NUM-SEEDS green patches as "seeds"
    ask n-of num-seeds patches
       [ set pcolor green ]
    create-turtles num-particles
       [ set color red
           set size 1.5
           setxy random-xcor random-ycor ]
    reset-ticks
end
```

The N-OF reporter selects a random set of NUM-SEEDS patches. We then ask these randomly chosen patches to turn green, making them seeds for the aggregates. This is different from what we did in the Fire model; in that model, we asked each of the patches to determine if they should become a tree (a seed) using a random variable. The Fire model method will generate roughly the fraction of trees specified by the slider; the N-OF method will always generate exactly the number of seeds specified by the slider. In agent-based modeling, the probabilistic method used in the Fire model is often considered more realistic since nature does not specify exact numbers. However, the N-OF method gives us more precise control over the model's behavior.

This code works, but note that it no longer re-creates the exact results of the original model. Even if we set NUM-SEEDS to 1, we will no longer always create a seed at the center of the world. Instead, the seed will be randomly placed somewhere in the world. In this case it is simpler to deviate from the original version and not much is lost by doing so. There is really nothing special about the center of the world, and having the seed start anywhere randomly will be sufficiently similar to the original result for testing and exploration purposes. If we really desire to have the seed start at the center of the world, we could add another parameter; this is left as an exercise for the reader. The final version of the simple DLA model, after all three extensions, is shown in figure 3.13. With multiple seeds, the patterns look somewhat like frost forming on a cold window.

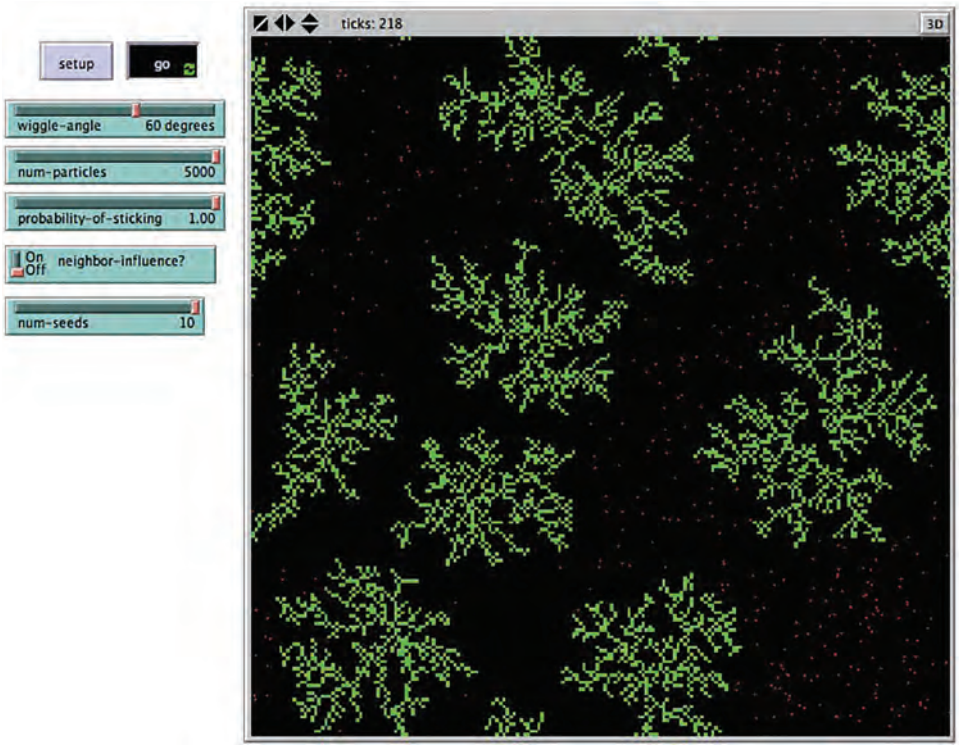

Figure 3.13
DLA model after third extension.

Summary of the DLA Model

In this section, we created three versions of the DLA model that enable us to generate different interesting patterns from very simple rules. Despite the complex and intricate nature of these patterns, the rules can be described in just a few words. The first two extensions modified how the particles "decide" when to stop moving. These are classical extensions and can result in thicker and more substantial patterns. The final extension added the idea of starting from multiple seeds, which enables the generation of different patterns by the same model and these different patterns can then be compared and contrasted side by side. The DLA model is a classic example of simplicity at a micro level creating complexity at a macro level.

Advanced Modeling Applications

The DLA model is one simple example of how ABM can be used to examine phenomena from classical chemistry and physics. There are many more examples in the NetLogo models library. For example, the Connected Chemistry (Wilensky, Levy & Novak, 2004)

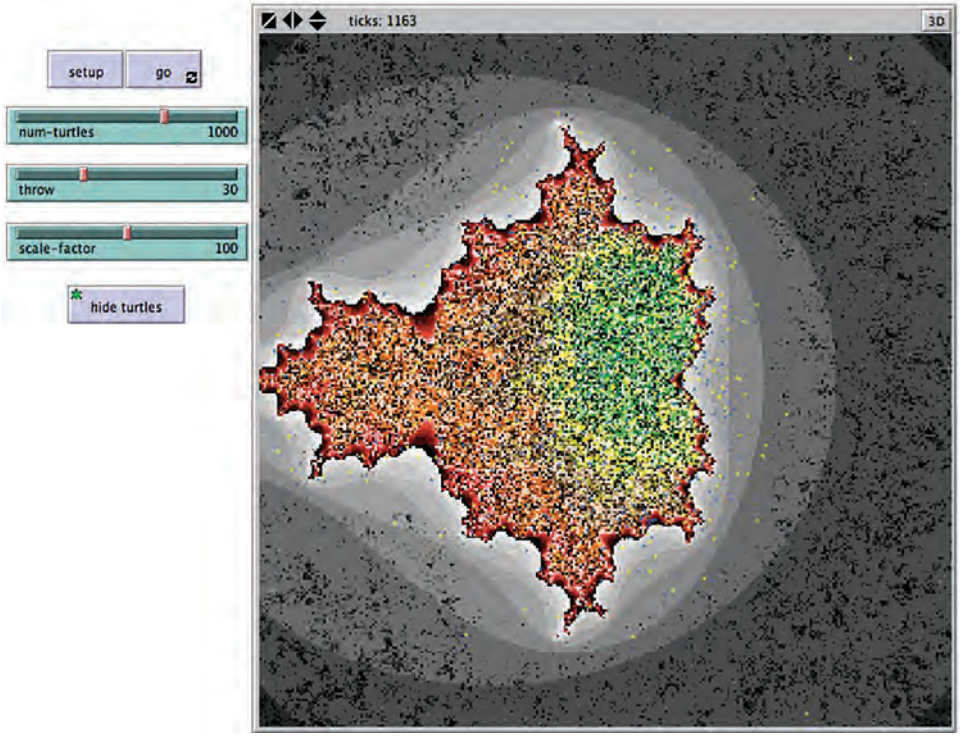

Figure 3.14
NetLogo Mandelbrot model. http://ccl.northwestern.edu/netlogo/models/Mandelbrot (*Wilensky, 1997c*).

model package models many standard chemistry principles using NetLogo, while NIELS (Sengupta & Wilensky, 2005) does the same for electromagnetism. The DLA model straddles the line between agent-based modeling and the mathematics of fractals. There are many other models in the NetLogo models library that involve the mathematics of fractals (see the Mathematics section of the models library in the subsection Fractals). Since many fractal systems are actually created using conditional rules, agent-based modeling is a natural way to describe these systems, and the results can be quite beautiful (see figure 3.14).

The Segregation Model

There are two fundamental approaches to starting to build an agent-based model. The first approach, more common in science contexts, is to start with a known phenomenon to be modeled. This approach is called *phenomena-based modeling.* Usually when engaged in phenomena-based modeling, we have an aggregate pattern, termed a *reference*

pattern that we are trying to generate with agent rules. Another way to begin is to start with some simple rules and play them out to see what patterns develop. This approach is sometimes referred to as *exploratory modeling*. Frequently in pursuing ABM, we take some combination of these two approaches. In his investigations into the nature of nature of segregation, Thomas Schelling took a more exploratory modeling approach with his Neighborhood Tipping model (1971). Schelling wondered what would happen if you assume that everyone in the world wanted to live in a place where at least a reasonable fraction of their neighbors were like themselves, i.e., they have an aversion to being an extreme minority (which he called weakly prejudiced; see Anas, 2002). He explored this model using a checkerboard as a grid, with pennies to represent one race and dimes to represent another. He counted by hand the number of pennies surrounding each dime and divide by the number of neighbors surrounding that dime. If this value exceeded a certain threshold value, he would move the dime to a random empty location on the board. He repeated this process hundreds of times and observed the outcome. (See figure 3.15.)

For high levels of the threshold, the model confirmed what he predicted. The checkerboard would quickly segregate into areas of all pennies and all dimes. What surprised Schelling about this model (and many other people at the time) was that even at low levels of the threshold, he would still see groups of pennies and groups of dimes together in dense clusters. This global segregation occurred despite the relative tolerance of each individual. There was no single individual in the Schelling model who wanted a segregated neighborhood, but the group as a whole moved toward segregation. Schelling described this as "macrobehavior" derived from "micromotives" (1978).

At the time when Schelling introduced this model, it was very controversial for several reasons. First, it was widely believed at the time that housing segregation was caused by individuals being prejudiced. Schelling's work seemed to be "excusing" people from prejudice, saying that prejudice itself is not the cause of segregation. The cause of segregation is an emergent effect of the aggregation of people's weak prejudice. But Schelling's point was not to excuse people for their prejudice, his model demonstrates that prejudice is not a "leverage point" of the housing system. Unless you can reduce prejudice to close to zero, so that everyone is perfectly comfortable being the sole member of their race in their neighborhood, this segregation dynamic will emerge. So, if your goal is to reduce housing segregation, it may not be effective to work directly on reducing individual's prejudice.

Another controversial aspect of Schelling's model is that it modeled the people as behaving with very simple rules. Critics argued that "people are not ants"; they have complex cognition and social arrangements, and you cannot model them with such simple rules. To be sure, Schelling's model was highly oversimplified model of people's housing choices. Nevertheless, it did reveal a hitherto unknown important dynamic. Eventually, Schelling carried the day. For his large body of work on exploring the relations of

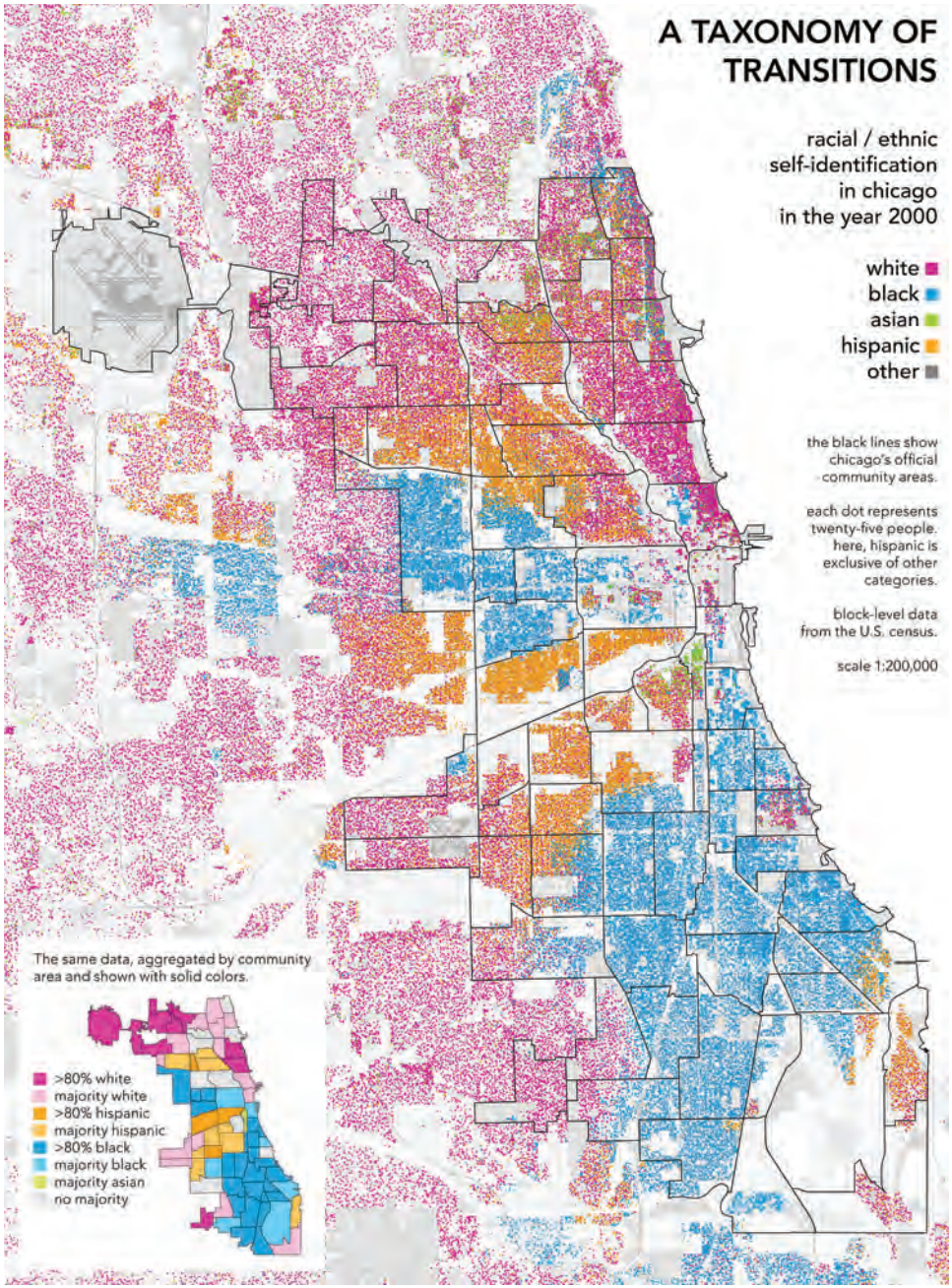

Figure 3.15
A map of Chicago with each dot representing twenty-five people. By representing the data at this finer level than traditional maps that aggregate demographics are mapped at a high level, Rankin illustrates nuances of segregation much like ABM enables us to explore nuances of individual-level behavior.

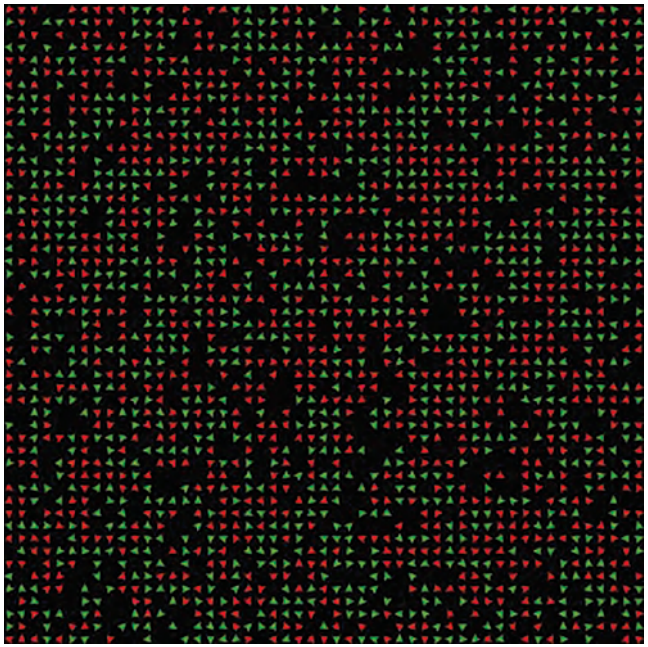

Figure 3.16
Initial state of the NetLogo Segregation model. Red and green turtles are distributed at random. http://ccl.northwestern.edu/netlogo/models/Segregation (*Wilensky, 1997d*).

micromotives and macrobehavior and his work on conducting game-theory analyses of conflict and cooperation, Schelling was awarded the Nobel Prize in economics in 2005. To this day, it remains controversial how much of human behavior can be modeled with simple rules. We will explore this further in later chapters of this textbook.

Description of the Segregation Model

A NetLogo version of the segregation model is distributed in the IABM Textbook section of the models library. You can find it in Sample Models > IABM Textbook > Chapter Three > Segregation Extensions > Segregation Simple.nlogo. This model is shown in figure 3.16. When you press the SETUP button, an approximately equal number of red and green turtles will appear at random locations in the world.

Each turtle will determine if it is happy or not, based on whether the percentage of neighbors that are the same color as itself meets the %-SIMILAR-WANTED threshold. When you press the GO button, the model will check if there are any unhappy turtles. Each unhappy turtle will move to a new location. It moves by turning a random amount and moving forward a random amount from 0 to 10. If the new location is not occupied then the turtle settles down, if it is occupied, it moves again. After all the unhappy turtles

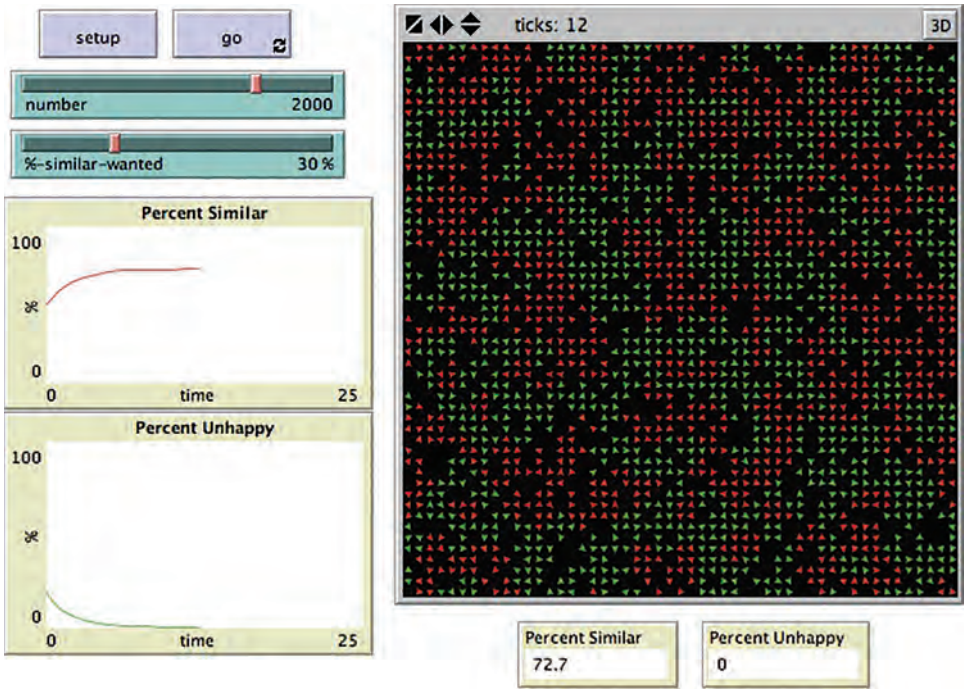

Figure 3.17
A run of the Segregation model with a "tolerance" level of 30 percent. Even though most agents are comfortable in an integrated neighborhood, the housing gets very segregated with 72.7 percent of agents surrounded by their same color agents.

have moved, every turtle again determines whether or not it is happy and the process repeats itself. (See figure 3.17.)

The SETUP procedure for this model is as follows:

```
to setup
    clear-all
    ;; create a turtle on NUMBER randomly selected patches.
    ask n-of number patches
       [ sprout 1 ]

    ask turtles [
       ;; make approximately half the turtles red and the other half green
       set color one-of [red green]
    ]

    update-variables ;; update the turtles and the global variables
    reset-ticks
end
```

The code asks a random set of patches to each sprout a red turtle. The size of the patch set is determined by the NUMBER slider, so after sprouting, the number of turtles is exactly the value on the NUMBER slider, each turtle on its own patch. The code then asks the turtles to turn red or green with equal probability, resulting in an approximately equal number of red and green turtles.

The GO procedure is:

```
to go
    if all? turtles [happy?] [ stop ]
    move-unhappy-turtles
    update-variables
tick
end
```

This code stops the simulation when all the turtles are happy. If there are any unhappy turtles, it asks them to move, and then all turtles recalculate their happiness.

The happiness calculation takes place in the update-turtles procedure, which follows:

```
to update-turtles
    ask turtles [
       ;; in next two lines, we use "neighbors" to test the eight patches
       ;; surrounding the current patch

       ;; count the number of my neighbors that are the same color as me
       set similar-nearby count (turtles-on neighbors)
          with [color = [color] of myself]

       ;; count the total number of neighbors
       set total-nearby count (turtles-on neighbors)

       ;; I'm happy if there are at least the minimal number of
       ;;    same-colored neighbors
       set happy? similar-nearby >= ( %-similar-wanted * total-nearby / 100 )
    ]
end
```

This code asks each turtle to count how many of its neighbors are same-colored turtles and how many are differently colored turtles. It then sets its "happy?" variable to TRUE if the percentage of same-colored neighboring turtles is at least equal to the value set by the %-SIMILAR-WANTED slider.

The PERCENT SIMILAR monitor displays what percentage of a turtle's neighbors are the same color as it, on average, while the PERCENT UNHAPPY monitor display what percentage of all turtles are unhappy.

First Extension: Adding Multiple Ethnicities

One simple extension to this model is to add a third, fourth, and even fifth ethnicity. This can be done by modifying the SETUP code. Currently the model sets all the turtles to red initially, then asks half of them to become green. Recall the code that sets up the turtles:

```
;; create turtles on random patches
ask n-of number patches
   [ sprout 1 ]
ask turtles [
   ;; make half the turtles red and the other half green
   set color one-of [red green]
]
```

We need to figure out how to modify this code so that it works with more than two ethnicities. Since we are now allowing the number of colors to vary we need to go beyond red and green turtles, sometimes we will need blue, yellow, and orange turtles as well. Therefore, we begin by defining the colors we will allow the turtles to have. We do this by establishing a global variable *colors*:

```
globals [
percent-similar    ;; on average, what percent of a turtle's neighbors
                                                ;; are the same color as that turtle?
    percent-unhappy  ;; the percent of the turtles that are unhappy
    colors                              ;; a list of colors we use to color the turtles
]
```

The last line, "colors," is the only new global variable. Now we need to define what the value of this global is. We can do this in the SETUP procedure:

```
    set colors [red green yellow blue orange ]
```

This line of code initializes the global variable *colors* to be a list of the five colors. Next, we would like to allow the model user to control of the number of ethnicities in the system, and we can do this by adding a NUMBER-OF-ETHNICITIES slider. We will initially set the bounds of this slider from 2 to 5. Now we have to make the code use this slider. We do this by modifying the turtle coloring code. We replace all of the SETUP code above with the following:

```
;; create a turtle on NUMBER randomly selected patches.
         ask n-of number patches
                      [ sprout 1 ]
```

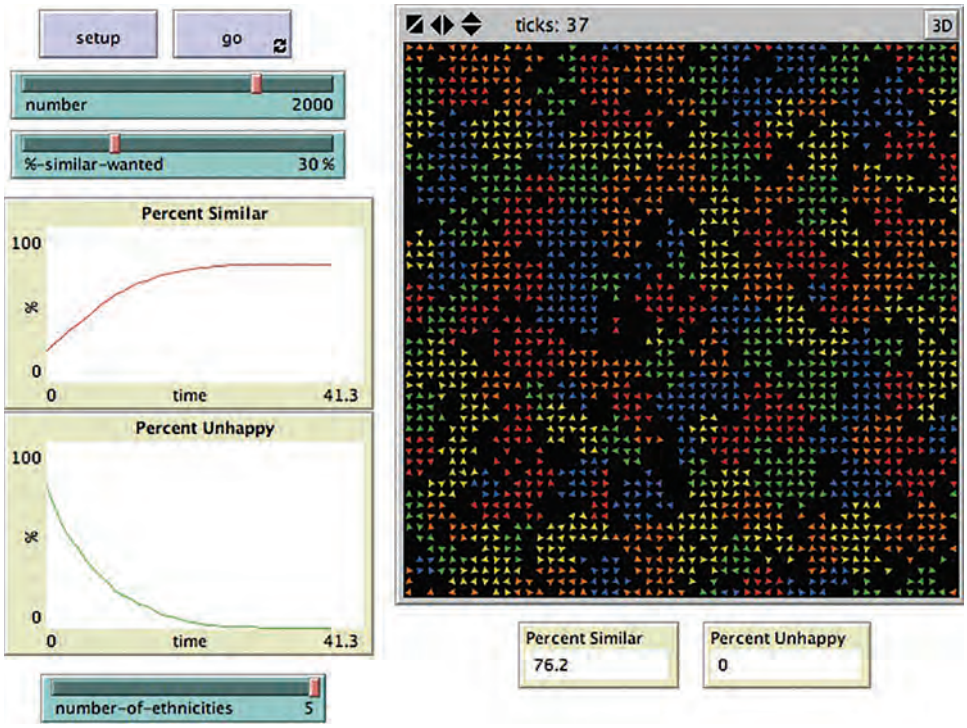

Figure 3.18
NetLogo Segregation model after first extension.

```
;; assign a color to each turtle from the list of our colors
        ask turtles
        [ set color (item (random number-of-ethnicities) colors) ]
```

This code tells a number (equal to the interface slider NUMBER) of random patches to sprout a turtle. Each turtle picks its color randomly from the list of colors we just initialized. But the turtles can only pick the colors in the list up to the number of ethnicities. For example, if the NUMBER-OF-ETHNICITIES is 3, only the first three colors in the list are used.

Now that we have code that allows multiple ethnicities, try running the model with different numbers of ethnicities and observe what effect this has on the segregation of the system. You will notice that as you increase the NUMBER-OF-ETHNICITIES, it takes longer for the system to settle—that is, for all the turtles to be happy. (See figure 3.18.) However, once the system has settled, the final PERCENT SIMILAR displayed in the monitor remains pretty much the same regardless of how many ethnicities there are. Can you explain why this might be?

Second Extension: Allowing Diverse Thresholds

In the original version of Schelling's model, every agent in the world had exactly the same similarity threshold. This meant that every agent had the same level of tolerance for having other ethnicities in its neighborhood. Although relaxing this assumption might have been difficult to enact with a checkerboard, it is straightforward to do using an ABM language. It's likely that different individuals in a real population would have different levels of tolerance. And with an ABM language we can easily give each individual agent its own personal characteristics. Agents can act on these personal characteristics and make decisions based on them. This means that even when agents in an ABM are running the exact same code, their behavior can be quite different.

So how can we make the individuals in the Segregation model more diverse? One way would be to give them a range of similarity thresholds rather than requiring them all to have the same value for that threshold. To do this, we need to first give the turtles the ability to have their own %-SIMILAR-WANTED. We do this by modifying the *turtles-own* properties. As we showed in chapter 2, any variable in a turtles-own declaration is a variable that is a property of every turtle, that means every turtle can have a different value for that variable.

```
turtles-own [
    happy?          ;; for each turtle, indicates whether at least
    ;; %-SIMILAR-WANTED percent of that turtle's neighbors
    ;; are the same color as the turtle
    similar-nearby  ;; how many neighboring patches have a turtle with
           ;;  my color?
    total-nearby  ;; sum of previous two variables
    my-%-similar-wanted       ;; the threshold for this particular turtle
]
```

Now, in addition to the other properties that the turtles have, they each have a MY-%-SIMILAR-WANTED property, but as of right now this value is not set. We need to modify the SETUP procedure in order to initialize the turtles with their own MY-%-SIMILAR-WANTED values.

```
;; assign a color to each turtle from the list of our colors using number-of-
;; ethnicities to maximize the number of colors and assign an
;; individual level of %-similar-wanted
    ask turtles
        [ set color (item (random number-of-ethnicities) colors)
          set my-%-similar-wanted random %-similar-wanted[5] ]
```

5. Note that by making this modeling choice, we have made it impossible to reproduce the model behavior before the extension. In general, when extending models, it is inadvisable to eliminate previous behaviors. In this case, we did it for clarity of exposition. To make this extension "backward compatible," you could use the "random-normal primitive, and then to reproduce the previous behavior, you would give it a standard deviation of 0.

We insert this new code directly after specifying the color that we changed in the first extension. This value is set to a random value between 0 and the value from the %-SIMILAR-WANTED slider. This means that the %-SIMILIAR-WANTED variable no longer specifies the tolerance of each agent, but rather it specifies a maximum value that any agent can have for its tolerance.

However, the code that determines whether or not a turtle is happy has not yet changed, and so all of these individual tolerance values will be ignored. Let us change the UPDATE-TURTLES code to use these new values. This requires only one small change to the code:

```
set happy? similar-nearby >= ( my-%-similar-wanted * total-nearby / 100 )
```

All we do is replace %-SIMILAR-WANTED with MY-%-SIMILAR-WANTED to tell the turtle to use its own agent variable instead of the global variable.

Now that we have all of the code for the agents to use their own thresholds, run the model several times with different values of %-SIMILAR-WANTED. Do you see any different results? If you play with the model enough, you will notice that the PERCENT SIMILAR monitor ends up with a lower value at the end of the run then it did after the first extension. This result is logical, since no individual can be less tolerant in this extension than before, but some individuals will be more tolerant allowing them to find locations where they are happy even though there are more ethnically different individuals around them. In fact, the average of the MY-%-SIMILAR-WANTED values of the turtles will be approximately half of the global %-SIMILAR-WANTED slider. Comparing the original Segregation model with this extension of the model using twice the %-SIMILAR-WANTED value is left as further exploration for interested readers. (See figure 3.19.)

Third Extension: Adding Diversity-Seeking Individuals

Another simplification that the original segregation model makes about the world is that individuals are concerned about too much diversity but that no agents actively seek diversity in their neighborhoods. An easy way to overcome this simplification is to allow individuals to have a %-DIFFERENT-WANTED property just as they have a %-SIMILAR-WANTED property. We begin by creating a slider for %-DIFFERENT-WANTED that is just like the %-SIMILAR-WANTED slider. After that we only need to make one additional change to the code. We've already been calculating the number of ethnically different turtles in the neighborhood and so we only have to modify the update-turtles procedure to make use of the new parameter:

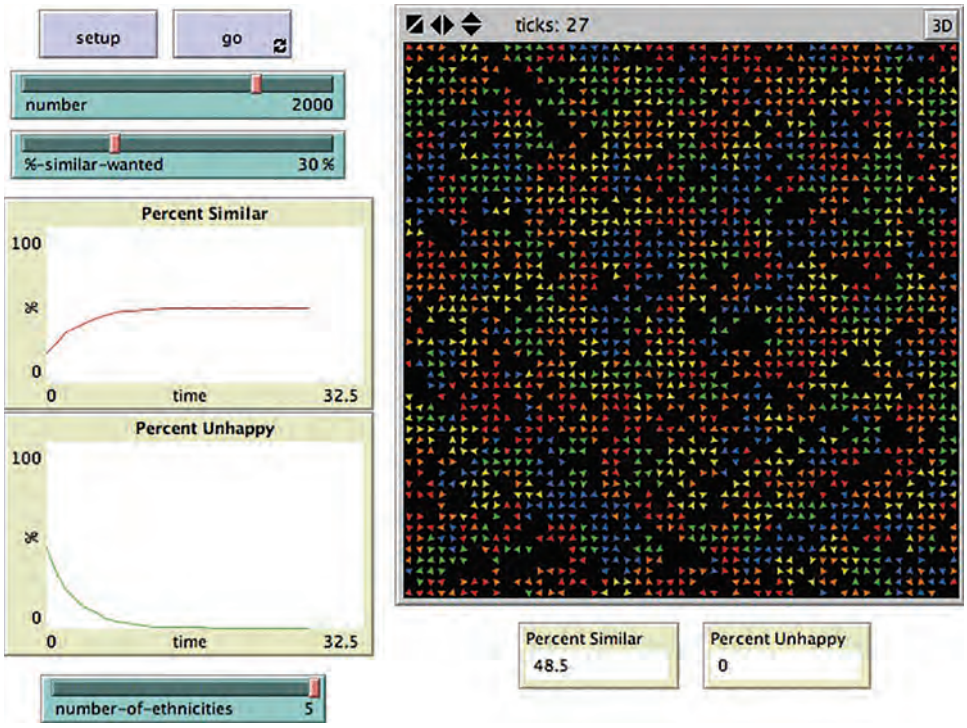

Figure 3.19
NetLogo Segregation model after second extension.

```
;; count the number of my neighbors that are a different color than me
let other-nearby count (turtles-on neighbors)
          with [color != [color] of myself]

set happy? similar-nearby >= ( my-%-similar-wanted * total-nearby / 100 )
                 and other-nearby >= ( %-different-wanted * total-nearby / 100 )
```

With the addition of the new clause (shown in bold), the new happiness calculation tells the agents that they are only happy if the number of similar agents nearby is greater than their %-SIMILAR-WANTED threshold *and* the number of other agents nearby is greater than the %-DIFFERENT-WANTED threshold. So for an agent to be happy it cannot be too much in a minority or too great a majority. Now that we have added this code to the model, run it a few times and observe the results.

As you play around with these sliders, it becomes quickly clear that it is much easier for the PERCENT SIMILAR results to decrease after this extension has been implemented. The reason is that all of the agents actually seek out diversity now, so they are deliberately taking actions that decrease the PERCENT SIMILAR results. Moreover, as you

manipulate the sliders, you can easily find states of the parameters where the system never settles down. For instance, if you put both the %-SIMILAR-WANTED and %-DIFFERENT-WANTED sliders over 50 percent, the model will probably never reach equilibrium. This is because some agents (those with MY-%-SIMILAR-WANTED greater than 50) will be trying to satisfy impossible demands; they will never find a location where more than half the agents around them are similar to themselves and more than half the agents around them are different than themselves.[6] In general you will find that it takes this new system much longer to settle down to an equilibrium state. This is because the agents are now pickier in terms of where they are happy. They are seeking both diversity and similarity. (See figure 3.20.)

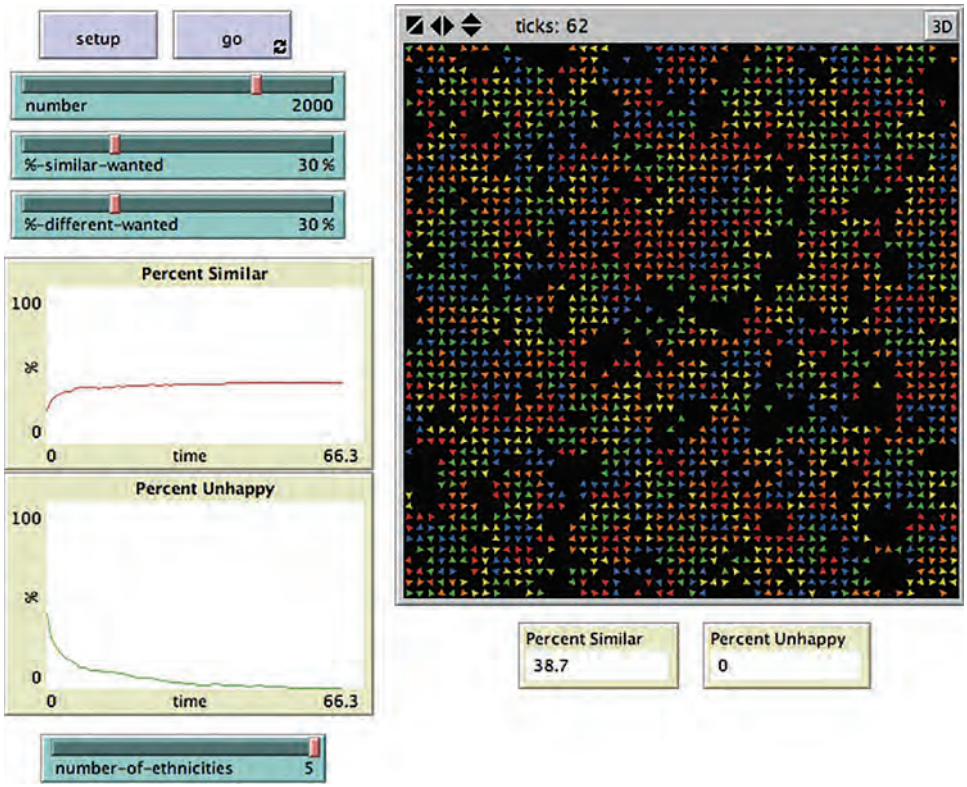

Figure 3.20
NetLogo Segregation model after third extension.

6. Contrary to the traditional assumptions in economics of a rational *Homo economicus*, people sometimes *do* have inconsistent preferences, and ABMs are well suited for modeling these inconsistencies.

There are two quick modifications that could be built to further expand on this model. First, the current %-DIFFERENT-WANTED slider specifies a global variable. It could easily be modified to give agents varying thresholds like we did for the %-SIMILAR-WANTED slider in the second extension. This would involve adding a MY-%-DIFFERENT-WANTED for each turtle, the same way we added MY-%-SIMILAR-WANTED. Second, we could modify the model so that some agents sought only diversity and some agents sought only similarity. How would you go about doing this? What effect would these two modifications have on the results?

Summary of the Segregation Model

One of the major aspects of ABM to take away from the Segregation model is that different models emphasize different aspects of the world. In the original model, Schelling made a particular set of assumptions about people's preferences and behavior. In the first extension we decided to place more emphasis on the multiplicity of ethnicities in the world than Schelling did. In the second extension, we shifted emphasis from uniform agents to heterogeneous agents who had different thresholds. This enables us to focus more on individual differences and diversity rather than on individual similarities and universality. Finally, in the third extension, we removed the emphasis on ethnocentric behavior that existed in Schelling's original model and instead started to examine agents that actually seek out diversity. Typically, ABMs can be built to emphasize particular features of the world or particular mechanisms of interaction. Choosing which features and mechanisms to select and trying out alternative selections is a part of the art of modeling complex systems.

Advanced Urban Modeling Applications

Agent-based modeling has frequently been used for the modeling of urban landscapes. The Segregation model described earlier can be seen as one particular example of what are generally called "residential preference" models, which are a major component of urban modeling systems. These models can be combined with commercial, industrial, and governmental models of urban policy to create an integrated model of a city. One such example is the CITIES project (see figure 3.21) carried out by the Center for Connected Learning and Computer-Based Modeling at Northwestern University. The goal of the CITIES project was to procedurally develop realistic cities that could be used as tools in education and exploration (Lechner et al., 2006). Another goal of urban modeling is to explore the ecological impact of cities and their footprint on the environment. The SLUCE project at the University of Michigan built several models of residential and tract development to explore land-use policies and their effect on the surrounding environs in southeastern Michigan (Brown et al., 2005; Rand et al., 2003). Both of these projects can be seen as a subset of the larger category of land-use modeling. For a detailed review of agent-based models of land use, see Parker et al. (2003) and Batty (2005).

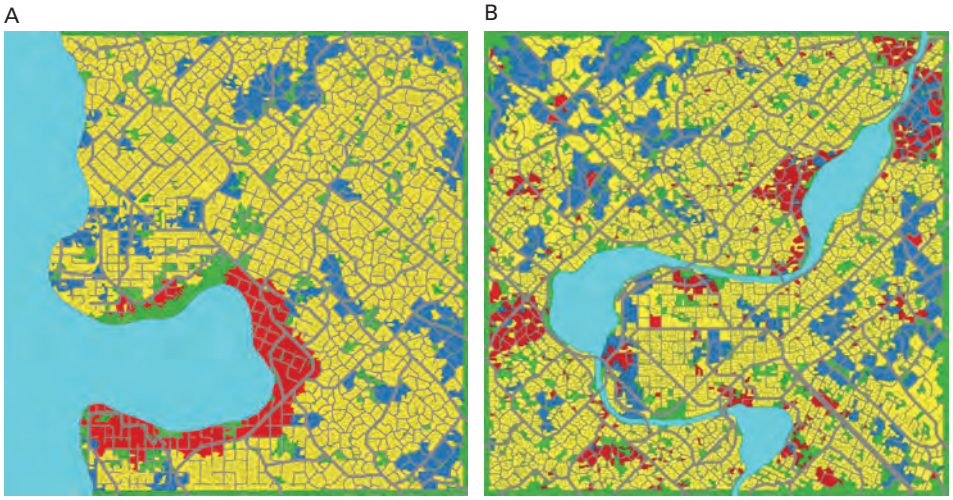

Figure 3.21
Some runs of the CITIES model in NetLogo (Lechner et al., 2006).

The El Farol Model

Occasionally, when working on agent-based models, you will find a model that reproduces phenomena that you are interested in but does not quite provide the output or data that you want. In this section we will take a model of an economic scenario, the El Farol model, and we will add additional reporters to this model. In this way we can gain new information from the model that was not previously available. This is possible because data collection in agent-based models is fairly simple, and the capabilities of ABMs for visually displaying information are diverse.

This section is a good example of how you do not need to understand every part of an agent-based model in order to work with it. The El Farol model uses "lists" and mathematical regression, but the modifications we will make to this model do not require knowledge of how these particular methods work. Although it is almost always desirable to understand the basic rules of the model, for the purposes of a particular extension, it may be sufficient to develop that understanding from reading the Info tab rather than trying to puzzle out the details of the code. When extending models, it is useful, though not easy, to develop the skill of understanding only the code you will need to modify and know which code is safe to ignore. Our extensions to this model will not require much understanding of the mechanics of the model.

Description of the El Farol Model

W. Brian Arthur was an Irish economist who, at age thirty-seven, became the youngest holder of an endowed chair at Stanford University. He was a pioneer of using complexity

methods in economics and is on the founders' board of the Santa Fe Institute. In 1991, Arthur posed the El Farol Bar problem as an exploration of bounded rationality and inductive reasoning. Traditional neoclassical economics presupposes that humans are completely rational: that is, they have access to perfect information and maximize their utility in every situation (Arthur, 1994). When each agent behaves this way the aggregate can achieve an "optimal" equilibrium. But this is an idealization and people do not really conform to this normative description. Therefore, Arthur suggested the El Farol Bar problem as an example of a system where agents do not perfectly optimize, yet a classical economic equilibrium is achieved. Arthur's example is even more striking, because it is a situation in which it would be difficult for the "ideal" economic model to achieve equilibrium.

On Canyon Road in Santa Fe, New Mexico, there is a bar called the El Farol. This bar was popular with researchers from the Santa Fe Institute, including Arthur. One night each week, the El Farol had live Irish music, and Arthur enjoyed going on these nights, but occasionally it got crowded, and he did not enjoy it on those nights. Arthur wondered, how do people decide if they should go to the bar or not? He imagined that there were one hundred citizens of Santa Fe who liked Irish music, and that each week they each tried to predict if the El Farol would be crowded. If they thought it was going to be crowded, specifically that more than sixty people would go to the bar, then they would stay at home, but if they thought it was not going to be crowded, then they would go to the El Farol. Assuming that the attendance information at the bar each week was readily available, but that each citizen could only remember a limited number of weeks of the attendance, Arthur hypothesized that one way to model this situation would be to give each agent a bag of strategies. Each of these strategies would be some rule of thumb about what the attendance was that week, e.g., "twice last week's attendance," "half the attendance of two weeks ago," or "an average of the last three weeks attendance." Each agent had a group of these strategies, and he would see how well these strategies would work had he used them in the previous weeks. The agents would use whichever strategy would have worked the best to predict this coming week's attendance, and they would decide whether to attend the bar based on this prediction. When Arthur wrote this up as an agent-based model and examined the results, he found that the average attendance at the bar was around 60. So despite all of the agents using different strategies and not having perfect information, they managed to optimally utilize the bar as a resource.

This model is available in the NetLogo models library under Sample Models > IABM Textbook > Chapter Three > El Farol (*Rand & Wilensky, 2007*; http://ccl.northwestern.edu/netlogo/models/ELFarol). This model has a few controls. You can adjust the MEMORY-SIZE of the agents, which controls how many weeks of attendance they remember. You can also change the NUMBER-STRATEGIES, which is the number of strategies each agent has in its bag of strategies. Finally, you can adjust the OVERCROWDING-THRESHOLD, which is the number of agents needed to make the bar crowded. If you

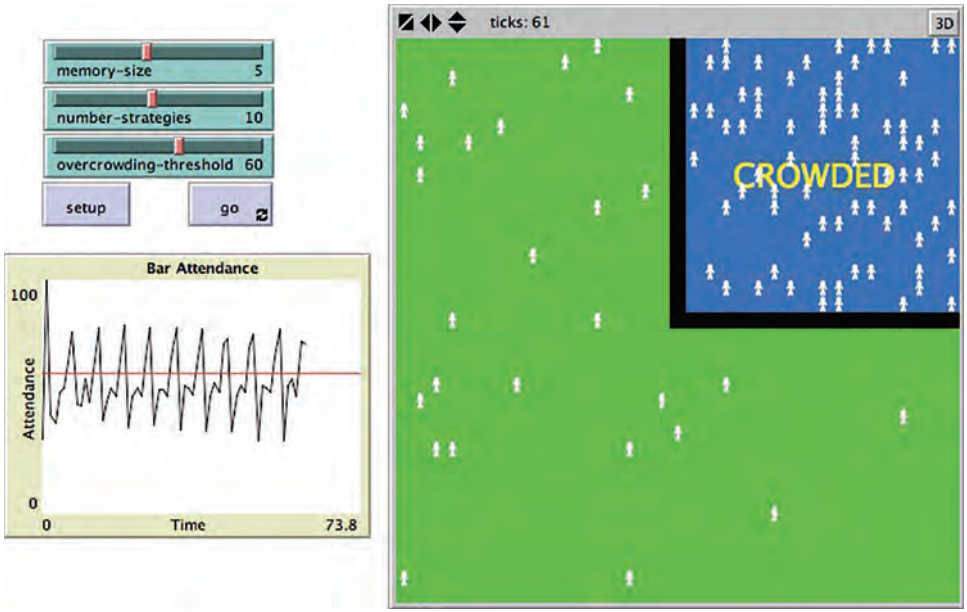

Figure 3.22
The El Farol model. Rand & Wilensky (2007). http://ccl.northwestern.edu/netlogo/models/ElFarol.

run the model as is, you will see the agents moving back and forth from the bar (the blue area) to their homes (green area), and the attendance is plotted on the left. The Interface tab of the model can be seen in figure 3.22.

First Extension: Color Agents That Are More Successful Predictors

The original model tells you how many agents are attending the bar, and you can get the visual reference of them moving to and from the bar, but there is nothing in this model to indicate which agents are better at choosing good times to attend the bar. Are some agents better at deciding when the bar will not be crowded than others? In order to find this out, we first need each agent to keep track of how often they go to the bar when it is not crowded; to do this, we add a property to each agent. We will modify the turtles' properties to add one additional item, REWARD:

```
turtles-own [
    strategies          ;; list of strategies
    best-strategy ;; index of the current best strategy
    attend?                ;; true if the agent currently plans to attend the bar
    prediction          ;; current prediction of the bar attendance
    reward                 ;; the amount that each agent has been rewarded
]
```

We are calling this new property REWARD, since the agents get rewarded if they attend the bar when it is not crowded. Now we have to initialize this property, since each agent will start with reward of 0. To do this we modify the create turtles part of the SETUP procedure, to initialize the REWARD to 0:

```
;; create the agents and give them random strategies
create-turtles 100 [
    set color white
    move-to-empty-one-of home-patches
    set strategies n-values number-strategies [random-strategy]
    set best-strategy first strategies
    set reward 0
    update-strategies
]
```

We also need to update the reward whenever the agent goes to the bar and it is not crowded. If we look at the GO procedure there is some code that determines if the bar is overcrowded or not and, if it is, the word CROWDED is displayed on a patch in the bar:

```
if attendance > overcrowding-threshold [
    ask crowded-patch [ set plabel "CROWDED" ] ;; label the bar as crowded
]
```

We want to update the reward of turtles who attend the bar when this condition is not true, so we can make this IF statement, an IFELSE statement, and then in the ELSE part of the IFELSE we can ask all the turtles who attended the bar to increase their reward:

```
ifelse attendance > overcrowding-threshold [
    ask crowded-patch [ set plabel "CROWDED" ]
]
[   ;; if the bar is not overcrowded, reward the turtles that are attending
    ask turtles with [ attend? ] [
        set reward reward + 1
    ]
]
```

Now that we have given rewards to agents attending the bar when it is uncrowded, the final step is to modify the visualization to display these rewards appropriately. One way to do that is to give each agent a different color based on how much reward it has accumulated. To make sure that we can keep track of relative differences in the model, we are going to color the agents relative to the maximum reward obtained. This means that we need to recolor every agent not just the ones that have gathered new rewards this time step.

If you look farther up in the GO procedure you will find some code where we ask each agent to predict the attendance that week. This is a good place to have the agents update their color. NetLogo has a command called SCALE-COLOR that allow you to do this, we can set each agent's color to a shade of a color based on their reward. The SCALE-COLOR primitive takes four inputs: (1) a base color, which we will make red, (2) the variable that we are linking the color to, REWARD in this case, (3) a first range value, in this case we set it to slightly more than the maximum reward so far, and (4) a second range value, which we set to 0. If the first range value for SCALE-COLOR is less than the second one, then the larger the number of the linked variable the lighter the color will be. If the second range value is less than the first one (as it is in this case), then the large the number of the linked variable, the darker the color will be. In this model, we set the first value to be the larger of the two so that everyone starts as white, and then turns a darker color as they accumulate rewards. This makes the model more consistent with the basic version, where all the agents are white:

```
ask turtles [
        set prediction predict-attendance best-strategy sublist history 0 memory-size
        ;; set the Boolean variable
        set attend? (prediction <= overcrowding-threshold)
        set color scale-color red reward (max [ reward ] of turtles + 1) 0
    ]
```

In this case, the darkest agents will be the one with the lowest reward, and the lightest agents will be the ones with the most reward (see figure 3.23).

Second Extension: Average, Min, and Max Rewards

The first extension enables us to visually see which agents are doing better at going to the bar when it is not crowded. However, we do not have any hard numbers for the agents. We could simply inspect each agent by right clicking on it and selecting inspect agent. This would enable us to see the reward for each agent, but a better way might be to constantly display some information about the agents. We can do this by using monitors, which display values about the agent-based models.

To begin with we add a monitor to the Interface tab. In the first monitor, we can calculate the maximum reward collected by any agent. In the reporter area of the monitor editing window, we can put the following code snippet:

```
max [ reward ] of turtles
```

We can also give this monitor the name "Max Reward." After this we can add another monitor for the minimum reward with the following code:

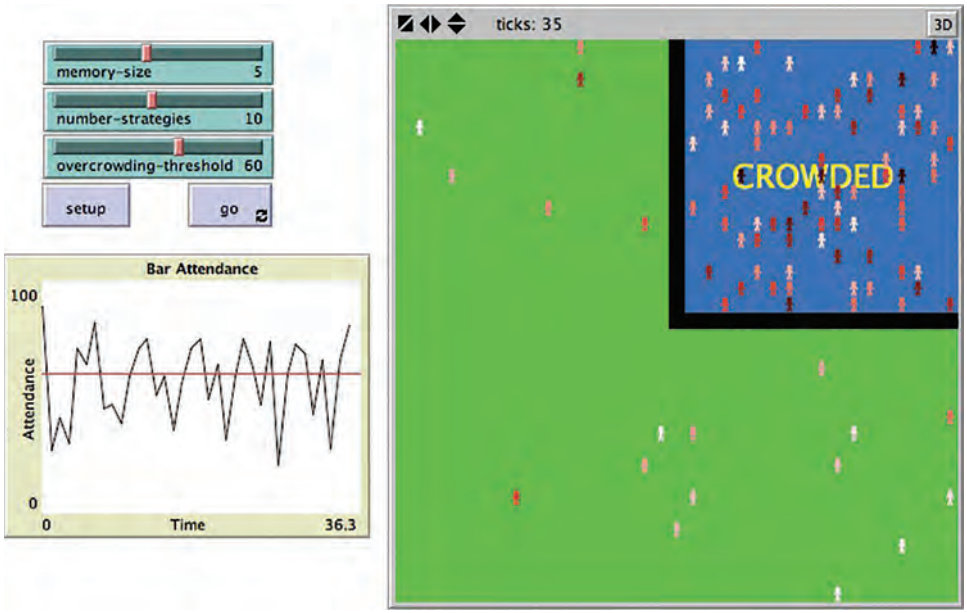

Figure 3.23
El Farol model: first extension.

```
min [ reward ] of turtles
```

And give this monitor the name "Min Reward." Finally, we can create a third monitor and calculate the average or mean reward using the following code:

```
mean [ reward ] of turtles
```

We can give this monitor the name "Avg. Reward." The model interface window now should look like figure 3.24. When we run this model we now get data on the average, maximum, and minimum rewards over time. It quickly becomes apparent that though both the average and maximum reward increase over time, the max reward grows faster than the average reward meaning that some agents do better and better as time goes on. Though this extension did not take much coding, it gives you information and insight into what is going on in the model that was not available before.

Third Extension: Histogram Reward Values

The second extension provides more data about how the agents in the model are doing, but this data has been aggregated so it is not obvious how many agents occupy each reward

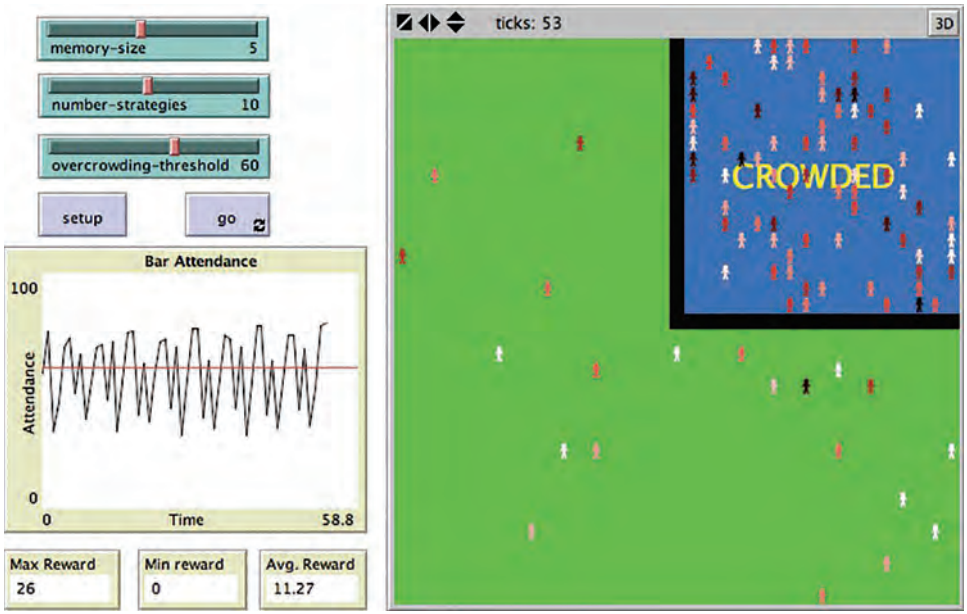

Figure 3.24
El Farol model: second extension.

level. For example, are the agents distributed uniformly across reward levels or are many agents at the minimum value and many agents at the maximum value with few agents in between? To find out an answer to this question, we are going to add another plot to the El Farol model. This plot will be a histogram of the distribution of rewards. This is similar to the plot we created in chapter 2 for the Simple Economy model, but in this model we will need to use the plot widget's "Plot update commands" feature.

To do this we first need to create a new plot in the Interface tab. In the configuration for this plot, we can give the plot the name "Reward Distribution," and change the Mode of the pen to "Bar" by clicking on the pencil in the first row of the Plot pens table. We tell the pen to display a histogram of the distribution by putting the following code in "Pen update commands."

```
histogram [ reward ] of turtles
```

This tells the pen to display a histogram of the REWARDs of the turtles. If we ran the model in this state, we would see a histogram drawn. However, the axes of the plot would not update correctly. Thus, we must enter the following under "Plot setup commands."

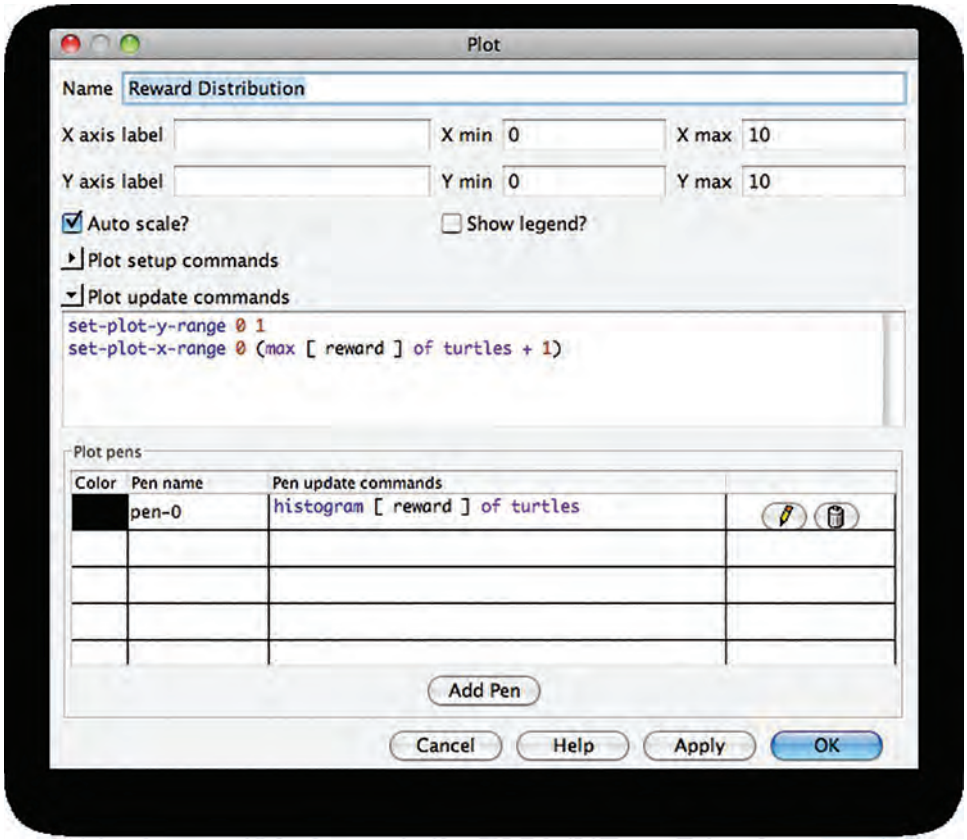

Figure 3.25
El Farol model: reward plot properties.

```
set-plot-y-range 0 1
set-plot-x-range 0 (max [ reward ] of turtles + 1)
```

These commands are run every time the plot is updated. They set the X and the Y range of the plot window to reflect the current values that we are plotting. By setting the Y-range to 1, we are letting NetLogo automatically decide how to increase this range if it needs to in order to show all the data. (See figure 3.25.)

As you run the model over time, you will see it start out similar to a normal distribution, but the distribution will quickly change with a few groups of individuals maintaining large rewards and many more having a lower reward. This lends credence to the hypothesis that there are a few agents that achieve a high reward level, but then there is a large gap between

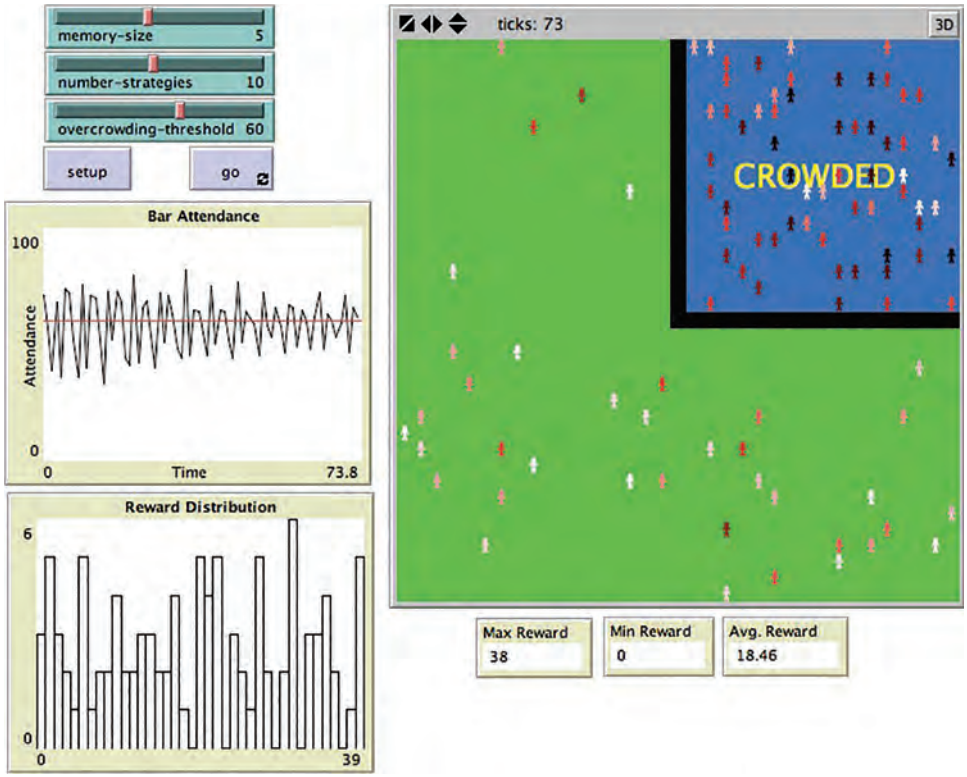

Figure 3.26
El Farol model: third extension.

these high achieving agents, and the majority of agents, which achieve more average reward levels. (See figure 3.26.)

Summary of the El Farol Model

In this section, we have discussed how we can modify the El Farol model so that it provides more data than the original version. In the first extension we provided an agent visualization that helped us to visualize the diversity in the success of the agents. In the second extension, we provided numerical output of the minimum, maximum and average reward values of the agent via monitors. Finally, in the third extension we created a histogram of that data which gives us a richer understanding of the underlying distribution if rewards per agent. These three output methods provide a richer view of how the El Farol Bar model works. If we had just been looking at the average attendance over time, it would not have become clear that some agents are doing very well and other agents are not.

A natural next question would be, "Why are some agents doing much better than others?" One thought would be that the agents that are doing well have a set of strategies that allow them to routinely outperform the other agents. We could look at these strategies and see how different they are from the strategies of agents that are not doing well. It may also be the case that agents who are doing extremely well are switching strategies more often or less often than agents who are not doing well, so that is another hypothesis that could be investigated. These further extensions are left as an exercise for the reader.

Advanced Modeling Applications

The El Farol model has been investigated in many different ways. Partially because it is an exciting model in that it combines both ABM and machine learning (Rand, 2006; Rand & Stonedahl, 2007). Machine learning enables the creation of powerful agent-based models where the agents not only change their actions over time, but also their strategies, i.e., the agents change the way they decide to take actions. We will discuss machine learning in more depth in chapter 5. The El Farol model has also been idealized into the Minority Game (Challet, Marsili & Zhang, 2004), which has been studied by physicists and economists because it provides insights into complex systems. For instance, both El Farol and the Minority Game can be seen as a crude approximation for financial markets. In some sense, if everyone else in a financial market is selling, you want to be buying, and if everyone else is buying then you want to sell; being in the minority is usually a good way to make money.

The El Farol model is also one example of an economic model. Another early agent-based economic model was the Artificial Stock Market developed by a group of Santa Fe Institute researchers (Arthur et al., 1997). This model attempted to simulate the stock market and allowed researchers to investigate how investors affect phenomena like booms and busts in the market. There are several other economic models in the NetLogo models library, such as the Oil Cartel model and the Root Beer Supply Chain (see figure 3.27).[7] Both of these models are interesting because they make use of the HubNet facility that is also provided with NetLogo. HubNet (*Wilensky & Stroup, 1999c*) is a NetLogo feature that enables Participatory Simulations, which is a simulation method where humans control agents in an agent-based model (see chapter 8 for more information on participatory simulations). This gives humans the ability to mix with nonhuman agents and to gain a deeper understanding of how their decisions affect and are constrained by a complex system.

After extending four NetLogo models, you should have the tools to extend many other NetLogo models. In the Fire model extensions and in the DLA extensions, we have seen that it is often useful to extend a model by replacing deterministic rules by probabilistic rules. Conversely, as we saw in modeling wind and long distance transmission in the Fire model, it is sometimes useful to replace a probabilistic rule with an agent-based mechanism that generates the probabilities. And with each change to the model, we need to consider

7. *Maroulis & Wilensky (2004)*. http://ccl.northwestern.edu/netlogo/models/HubNetOilCartel.

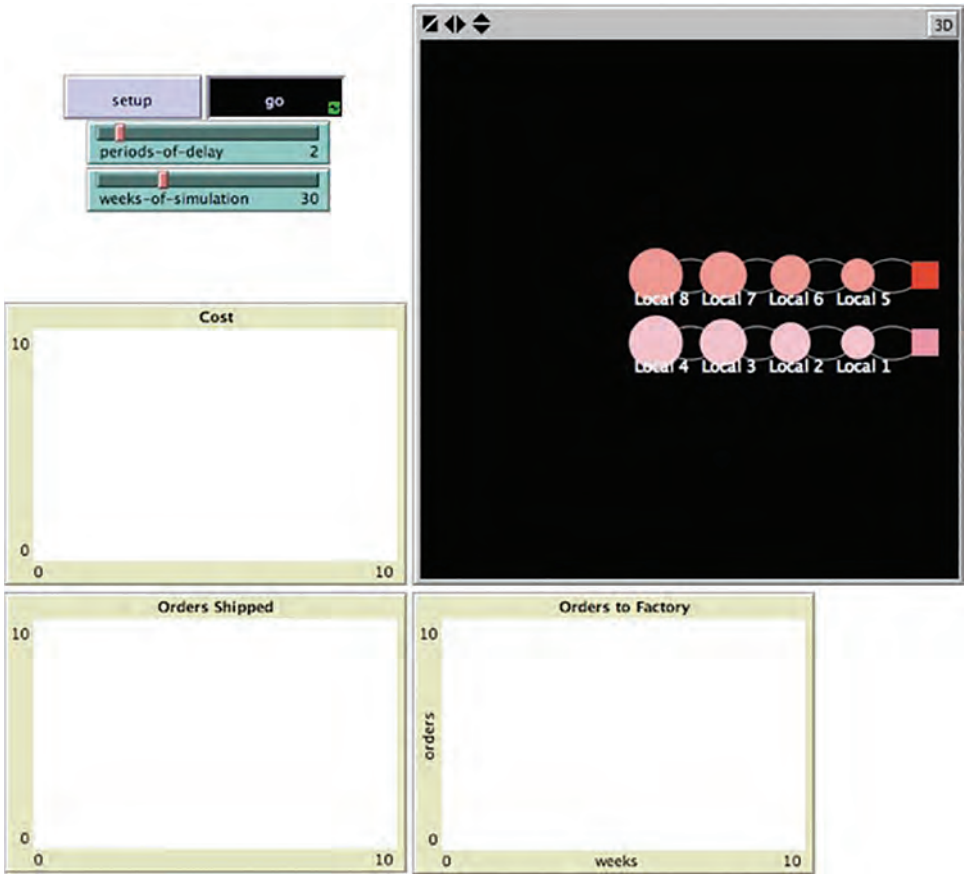

Figure 3.27
NetLogo HubNet Root Beer Game model. (*Wilensky & Stroup 2003*). http://ccl.northwestern.edu/netlogo/models/HubNetRootBeerGame.

whether we also need to design a new metric for measuring model performance. In the DLA model we saw that it can be useful to explore the effect of different rules. We also explored the effect of different starting and stopping conditions for that model. In the Segregation model, we explored changing global parameters, such as %-SIMILAR-WANTED to heterogeneous agent-level properties, and different possible numerations of the agent-classes, starting with Schelling's two ethnicities to multiple ethnicities. In addition, we looked at heterogeneity not just in agent properties, but also in agent-rules as some agents actively seek diversity. In the El Farol model we extended the model by having it provide us with more data, in the form of richer visualizations, numerical values, and information about the distribution of agent properties in the populations.

Conclusion

We have illustrated how to extend four different models, each in three different ways, and we have explored how these models relate to some of the key concepts of ABM. At this point you should have the tools and capabilities to start modifying models that you are interested in exploring on your own. You can go a long way in agent-based modeling using this approach. However, you may become interested in building a model that differs significantly from any extant model, and, in this case, the best solution will be to build your own new model from scratch. This is what we will discuss in the next chapter.

Explorations

1. Modify the Fire Simple model to use the fire-patches reporter.
2. Modify the Segregation Simple model to use two new reporters.
3. Write a reporter procedure for the Segregation model that reports the agentset of unhappy turtles. Write a second reporter that reports a statistic from the first reporter.
4. Write a reporter procedure that reports an agentset of all unhappy turtles. Then write another reporter that reports the average of number of similar neighbors for unhappy turtles. Can you use the first reporter as part of the code in your second reporter?
5. Write a reporter for the Simple Economy model from chapter 2 that reports when the wealth distribution reaches a stationary distribution.
6. We have seen that the Fire Simple model is based on a theory known as percolation that explains how less dense substances are able to make progress through more dense substances. One particular case where this is both interesting and potentially profitable is oil percolation. How can you change the Fire Simple model to be a model of oil percolation?
7. The Segregation model is overly simplified, partially because of the modeling resources that Schelling had access to at the time. Nonetheless, the model offers some strong lessons even though there are many mechanisms of housing selection that are left out. List three additional mechanisms/factors that could be included in the Segregation model, and give pros/cons for their inclusion. How do you think Schelling might have decided to include in the model and what not to include? When is it better to leave parts of a real world system out in order to emphasize other aspects?
8. *Ants with multiple pheromones* As mentioned in its Info tab, the Ants model focuses on the problem of food foraging, and assumes that ants can always find their own way directly back to the nest using a perfectly distributed predefined nest-scent gradient, once they have found the food. To make the model more physically plausible, change it so that the nest-scent is a new kind of pheromone that is released by ants for some limited time period after they have left the nest. This nest-scent pheromone should diffuse and evaporate just like the food-carrying pheromone does.

9. *Adding agent types* In the Disease Solo model in the Biology section of the NetLogo models library there is no way for an agent to recover after they have become infected with a disease. Add doctors to the model that wander around and cure the sick patients. How does this affect the results?

10. *Docking percolation and fire* Often when looking at the Fire model we ask what density of trees is necessary to reach the right side of the screen. We can ask the same question about the porosity value in the Percolation model and whether or not the oil in that model reaches the bottom of the screen. Modify the size of the world in the Percolation model so that it is similar to the Fire Simple model. What values of porosity do you need to get to the bottom of the screen? What values of tree density do you need to get to the right side of the screen in Fire? Are these values the same? If not can you explain why they might be different? If they are the same can you explain why that is?

11. *Making sparks probabilistic* Right now in the Fire Simple model as we extended it, sparks from fires always occur when a new fire is started. Modify this so that whether sparks are generated is probabilistic in the same way that the fire spread was made probabilistic in the first extension. How does this modify the behavior of the model?

12. *Random location of sparks* After the third extension in the Fire model, sparks are generated deterministically in the same place every time. This does not seem realistic, since real sparks probably jump to random nearby locations. Modify the Fire model so this is the case. How does this change the behavior of the model? Modify your model further by representing the sparks as turtles that can be visualized.

13. *Phase transitions and tipping points* Phase transitions and the related idea of tipping points occur in many different systems, where a small change in one parameter causes a large change in an output variable. The Fire Simple model (and percolation models in general) is an example of this. Name some other phenomena where phase transitions occur. Choose one of these phenomena and describe how you would build an agent-based model of this phenomenon.

14. In the original Fire Simple model, when the density is set to 50 percent, there are roughly an equal number of green patches (trees) and black patches (empty space). Many people guess that at that density the fire would have a good chance of spreading a lot. What stops the fire from spreading at this density?

15. In extension 1 of the Fire Simple model, there are two parameters governed by sliders, DENSITY and PROBABILITY-OF-SPREAD. How do these two interact with regard to a critical spread of the fire?

16. The original Fire Simple model has each patch check its four neighbors in the cardinal directions. Modify it so that it checks all eight of its Moore neighbors. How does this affect the spread of the fire? Is there still a critical density for the forest fire spread? If so, what is it?

17. In the Fire Simple model, the fire is initialized on the left edge of the view. Modify it so that it starts at a single random location. Can you redefine criticality under these

conditions? What is the critical threshold for spread under these conditions and with your definition of criticality?

18. Open the Sandpile model from the Chemistry and Physics section of the NetLogo models library. This model, originally developed by Bak, Teng, and Weisenfeld (1987), was the first model in which self-organizing criticality was presented. The white flashes help you distinguish successive avalanches. They also give you an idea of how big each avalanche was. Most avalanches are small. Occasionally a much larger one happens. Describe how it is possible that adding one grain of sand at a time can cause so many squares to be affected. Can you predict when a big avalanche is about to happen? What do you look for?

19. In the Sandpile model, the sand grains are distributed, one to each neighbor. Modify the model so that grains are distributed to neighbors randomly. How does this change the model behavior?

20. The Sandpile model exhibits characteristics commonly observed in complex natural systems, such as self-organized criticality, fractal geometry, 1/f noise, and power laws. These concepts are explained in more detail in Per Bak's book (1996). Add code to the model to measure these characteristics.

21. In the Sandpile model, try coloring each patch based on how big the avalanche would be if you dropped another grain on it. To do this, make use of the *push-n* and *pop-n* procedures so that you can get back to the distribution of grains before calculating the size of the avalanche.

22. In the last extension to the Segregation model we added a %-DIFFERENT-WANTED parameter that was a global variable for all agents. Modify this parameter so that each agent can have a different %-DIFFERENT-WANTED, similar to how we modified the %-SIMILAR-WANTED in the second extension. How does this change the results of the model?

23. In the Segregation model, the agents look to their neighborhood composition to decide if they are happy. Can you make the size of shape of the neighborhood a parameter? How does this change the results of the model?

24. In the Segregation model as we have extended it, agents can have different levels of %-SIMILAR-WANTED, and they also have a %-DIFFERENT-WANTED parameter, but these variables control all of the agents in similar ways. What if there are two kinds of agents, some who only seek diversity and some who only seek similarity? Modify this model so that this is the case. How does this change affect the results of the model?

25. *Modeling urban form* We have discussed how the Segregation model is one approach to modeling urban form. There are many different ABMs that capture various aspects of the creation of urban patterns. In NetLogo, the Urban Suite of models has several examples of this. Open the Path Dependence model in the Urban Suite and examine it. Modify the Path Dependence model to include elements of Schelling's segregation model. Give each

agent in the Path Dependence model a threshold of similarity and different colors in addition to its other properties. Have the agents move if they get unhappy with the distribution of colors in their current patch. How does this change the results of the original Path Dependence model?

26. The DLA Simple model often winds up with long, tendril-like particle traces. Why does this occur? Why does this pattern change when you make the decision to stick or not to stick probabilistically as we did in the second extension?

27. Modify the DLA Simple model so that it can have multiple colors of particles.

28. In the second extension of DLA Simple, we make the probability of sticking dependent on the number of neighbors. We accomplished this by multiplying the probability of sticking by the fraction of neighbors that are green. Is this modeling choice realistic for DLAs? Propose and defend another way of modeling the changing probability.

29. In El Farol Extension 1, we modify the color based on reward with the agents starting as white and becoming darker as they accumulate rewards. Can you modify this code to use a different color? Or can you modify it so the agents start out dark and become lighter over time? Is there some other visual way to indicate this reward?

30. In El Farol Extension 2, we display the Min./Max./Avg. Rewards, but these statistics are mainly used for a normal distribution. What should you display if you expected the rewards were not normally distributed? Modify the model to display some additional statistics about the reward distribution.

31. In El Farol Extension 3, we now graph the actual award distribution using a histogram. Another common way of displaying wealth distributions like this is to use a cumulative distribution function, where the y-value at any point is equal to the accumulation of the number of individuals with an x-value less than or equal to the current x-value. In other words, this graph would display how many agents have less than or equal to that much reward. Can you modify the model to add this graph instead of the histogram?

32. In the El Farol model, the weights that determine each strategy are randomly generated. Try altering the weights so that they only reflect a mix of the following agent strategies:

    (a) always predict the same as last week's attendance
    (b) an average of the last several week's attendance
    (c) the same as two weeks ago

    Can you think of other simple strategies that the agents could use?

33. Open the Rope model from the Chemistry and Physics section of the NetLogo models library. This model simulates a wave moving along a rope. The right end of the rope (shown in blue) is fixed to a wall. The left end of the rope (shown in green) provides an input, moving up and down in a sinusoidal motion. This creates a wave that travels along the rope. Change the right end of the rope so that it moves freely, rather than being fixed. How does that change the behavior of waves in the rope?

34. Open the Wandering Letters model from the Computer Science section of the NetLogo models library. This model illustrates how to build a word processor where each of the letters acts independently. Each letter knows only which letter comes before it and how long its word is. When the letters or margins are moved, the letters find their own ways back to their proper locations. Can you extend the model so the user can type his/her own message? You might want to use the user-input primitive for this.
35. Open the NetLogo Flocking model from the Biology section of the models library, which shows emergent rules for generating bird flocks. Can you add a predator bird that changes the other birds' behavior?
36. Choose any model from the Sample Models section of the models library and modify the rules for its agents. Please avoid the Games folder, the Optical Illusions folder, and the System Dynamics folder when you choose. If you choose Fire, DLA or Segregation, make sure your modifications are substantially different than any in the textbook. This should be a more substantial modification than the modifications we did in this chapter. Make sure that the modified model shows some interesting new emergent behavior. Write a short description of the rule changes and include a justification for why this model extension makes sense, and add it as a new section (EXTENSION JUSTIFICATION) to the model's Info tab. Describe how the behavior of the modified model differs from the original.
37. Create a simple model that includes some agents, a rule for birth of agents (see *hatch* primitive) and a rule for death of agents (see *die* primitive). It is often useful to include a rule for agent movement. The agents can be anything—animals, particles, organizations, etc. Can you find rules that have interesting behavior? Create a SETUP button to initialize your model and a GO button to run it. In your write-up of the model, make sure to *explicitly* describe the rules the agents are following. Try running the model in different ways. What set of parameters gives you the most interesting behavior? Use a text-based pseudo-code format to describe the rules.
38. In chapter 2, we looked at the cellular automata models Life Simple and Sample Models > Computer Science > Cellular Automata CA 1D Elementary. The first of these shows a simple 2D CA. The second shows a one-dimensional CA with its time evolution displayed in the second dimension. Create your own cellular automaton model. Decide on a dimension of the CA (If you like, you can use NetLogo 3D and explore a 3D CA). Decide on the radius of the CA neighborhood. Decide on either a synchronous or asynchronous update scheme. Give the CA at least three distinct states. Fill in a section of the info window that describes the rules for your CA. In another section of the info window, describe some interesting behavior of your CA.