3. Practice - NetLogo Simulations

Tamás Takács, PhD student, Department of Artificial Intelligence

90 min read

January 23, 2025

Collective Intelligence

Last Practice

In our last practice, we covered:

- An introduction to NetLogo and its competitor programmable modeling environments
- The advantages of using NetLogo
- Exploring the NetLogo environment:
  - World and Agents
  - Turtles and their properties
  - The Observer
- Core NetLogo components:
  - Reporters and Commands
  - Variables and their types
  - Conditionals and loops
- Advanced features:
  - Lists and higher-order functions
  - Tasks, including `map`, `filter`, and `reduce`
- Working with Breeds

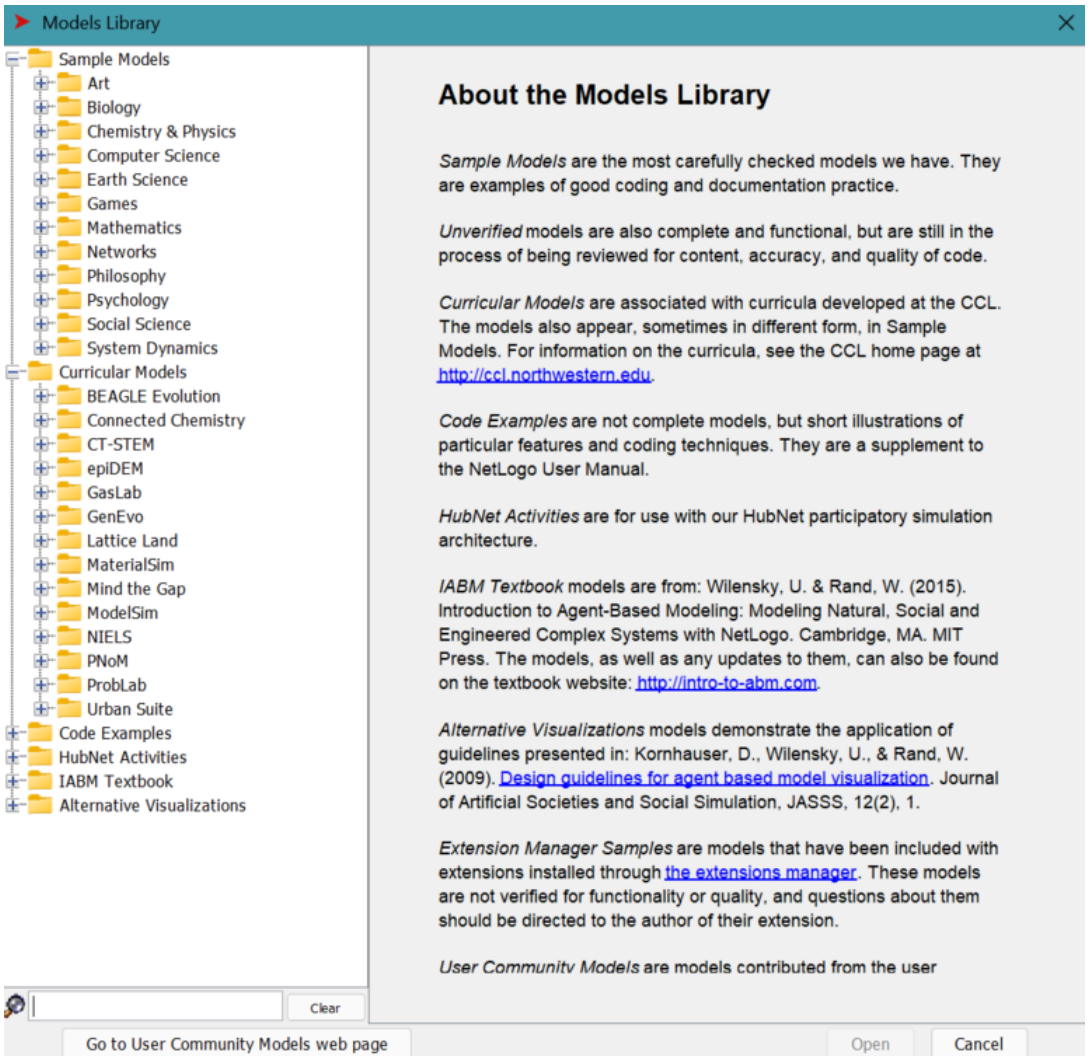
That's quite a lot! Now, let's dive into how these concepts are applied in simulations using the **Models Library**.

Models Library

The **NetLogo Models Library** is a comprehensive resource included with NetLogo, containing pre-built simulation models spanning diverse disciplines such as biology, economics, physics, and social sciences.

- It includes over **200 models**, categorized for easy navigation (e.g., Biology, Social Science, and Computer Science).
- Models like *Wolf Sheep Predation* and *Segregation* have become benchmarks for studying agent-based systems.
- Models are **fully editable**, allowing users to modify parameters, add features, or adapt them for custom research needs.

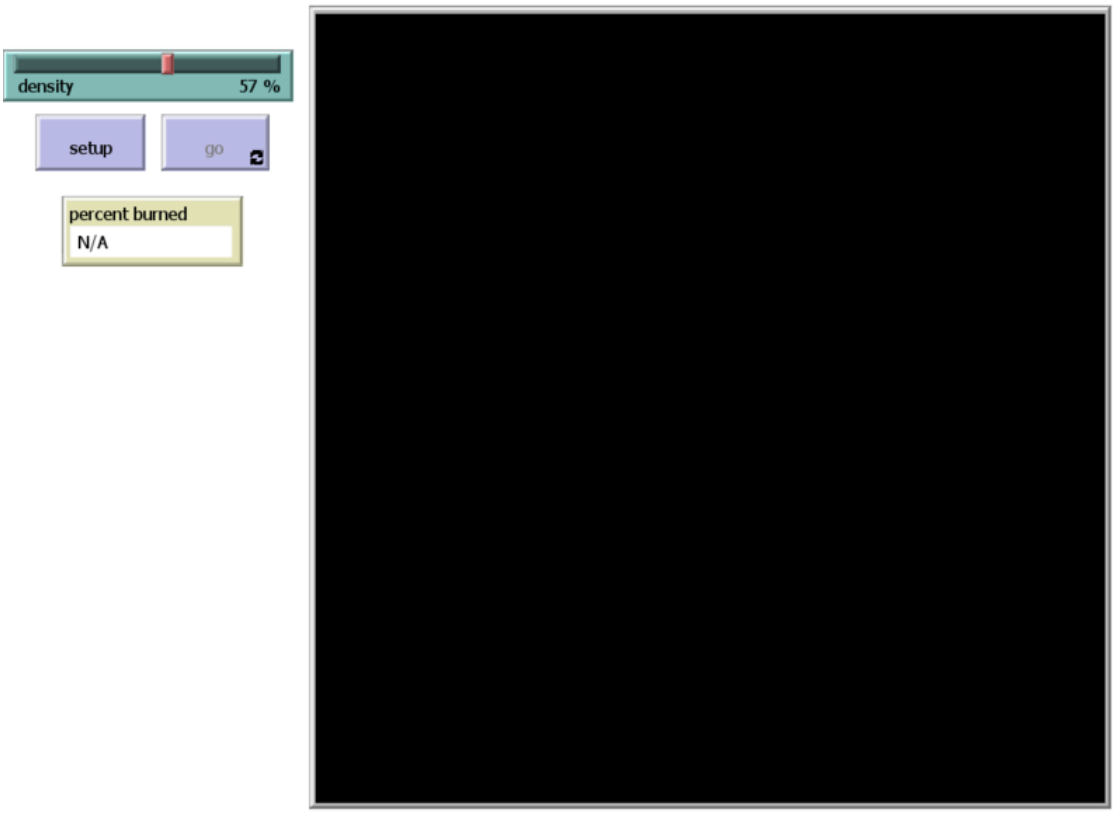
Benchmark Simulations



Your **Model Library** should look somehow like this under *File* > *Models Library.*

The Fire Model

File > *Models Library* > *Sample Models* > *Earth Science* > *Fire*



Interface Tab

The **Fire Model** is a simple yet illustrative agent-based simulation used to study the spread of fire through a forest, represented on a 2D grid. The environment consists of three primary elements:

1. **Trees**: Represented as green patches on the grid.
2. **Fires**: Represented by red turtles that simulate active fire.
3. **Embers**: Representing fading fire agents, marking burned areas.

Mechanics

The `setup` button initializes the forest grid based on a key variable called `density`. This global variable determines the proportion of the grid occupied by trees and is controlled by a slider in the interface. The `density` variable is only utilized during the setup phase and remains constant throughout the simulation, even if the slider is adjusted during runtime.

The `go` button initiates the core simulation loop, which governs the spread of fire:

- Fire agents ignite adjacent trees based on predefined rules.
- Burned trees transition into embers, and fire agents disappear once their task is complete.

The model includes a monitor labeled `percent burned`, which reports the percentage of the grid area affected by fire. This value is calculated by a reporter function that dynamically tracks the burned areas during the simulation.

Running the Simulation

Let's run the simulation with the initial `density` value of **57%**.

Exercise: Run the `setup` command in the Interface tab and press the `go` button to start the simulation.

Observe what happens to the forest:

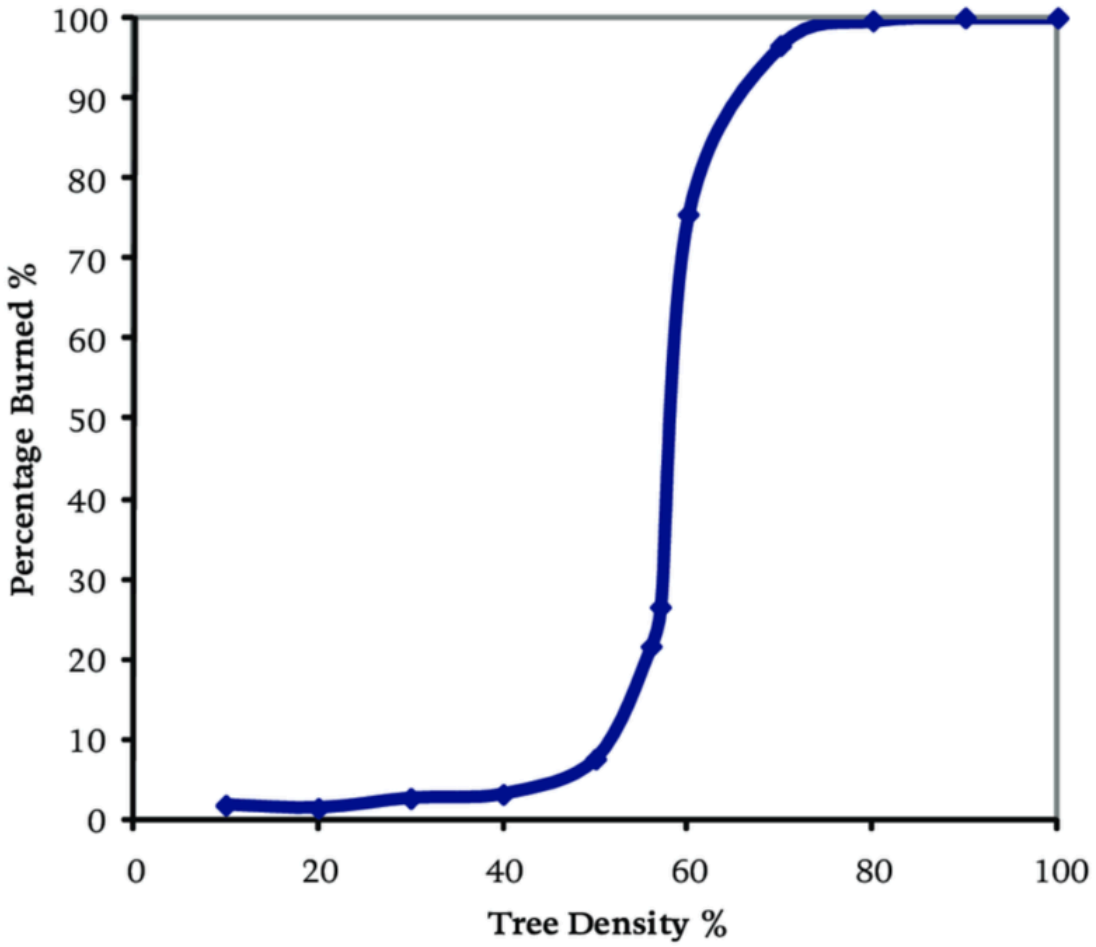
- Does the forest burn down completely? If yes, why? If no, why not?
- What is the **percent burned** value displayed in the monitor?
- Try to explain your observations based on the behavior of fire spread before looking into the code.

Exercise: Find the **epidemic threshold**—the critical tree density where the fire just barely spreads across the entire forest (agents reach the right side of the grid):

- Adjust tree density and observe when the fire reaches the right edge.
- Record the **percent burned** at densities above, below, and near the threshold.
- Explain why the fire behaves differently at these densities.

In modeling, this **epidemic threshold** is more accurately referred to as a **phase transition**—a point where a small change in an input parameter causes a dramatic shift in the system's behavior, leading to the collapse of a previously stable state.

For this simulation, there is a closed-form solution: with a tree density of 57%, there is nearly a 0% chance of the fire reaching the right side of the grid. However, as density increases to 62%, this probability jumps to nearly 100%.



Reference: Phase Transition Robertson, Duncan & Caldart, Adrián. (2008). Natural Science Models in Management: Opportunities and Challenges. E:CO Emergence: Complexity and Organization. 10.

The Code

Initial Setup

```
globals [
  initial-trees   ;; how many trees (green patches) we started with
  burned-trees    ;; how many have burned so far
]
```

```
breed [fires fire]    ;; bright red turtles -- the leading edge of the fire
breed [embers ember]  ;; turtles gradually fading from red to near black
```

```
to setup
  clear-all
  set-default-shape turtles "square"   ;; turn turtles into square shapes
  ask patches with [(random-float 100) < density]   ;; make some green trees
    [ set pcolor green ]

  ask patches with [pxcor = min-pxcor]   ;; make a column of burning trees
    [ ignite ]

  set initial-trees count patches with [pcolor = green]   ;; set tree counts
  set burned-trees 0
  reset-ticks
end
```

Helper Commands

```
;; creates the fire turtles
to ignite  ;; patch procedure
  sprout-fires 1
    [ set color red ]
  set pcolor black
  set burned-trees burned-trees + 1
end
```

```
;; achieve fading color effect for the fire as it burns
to fade-embers
  ask embers
    [ set color color - 0.3  ;; make red darker
      if color < red - 3.5     ;; are we almost at black?
        [ set pcolor color
          die ] ]  ;; Removes the turtle, only the black patch remains
end
```

Main Loop

```
to go
  if not any? turtles  ;; either fires or embers
    [ stop ]
  ask fires
    [ ask neighbors4 with [pcolor = green]
        [ ignite ]
      set breed embers ]
  fade-embers
  tick
end
```

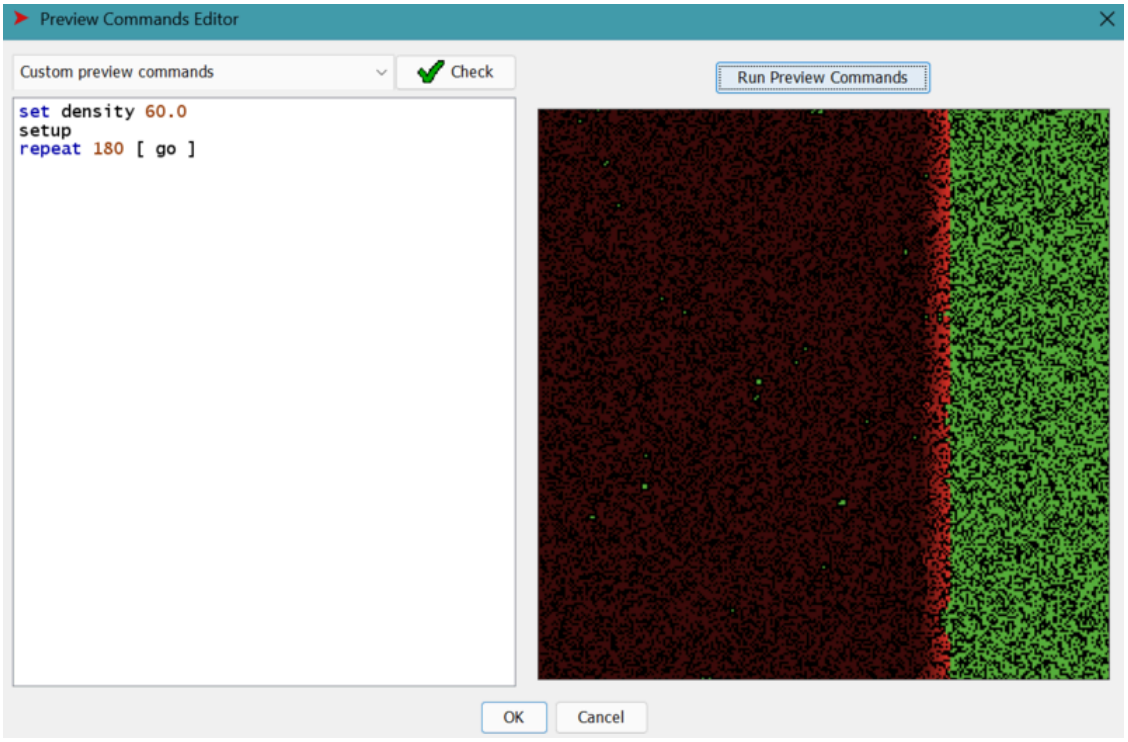
Exercise: Modify the code to allow the fire to spread to all 8 neighboring patches (not just the 4 directly adjacent ones).

Run the `setup` command in the Interface tab and press `go` to start the simulation.

- Observe how the fire spreads. Does it behave differently with the change?
- Does the forest burn down completely? If yes, why? If no, why not?
- Determine the new `density` for which a **phase transition** happens. How does it compare to the original density threshold?
- Do you think the critical density needs to be halved or adjusted differently? Explain why this might be the case.

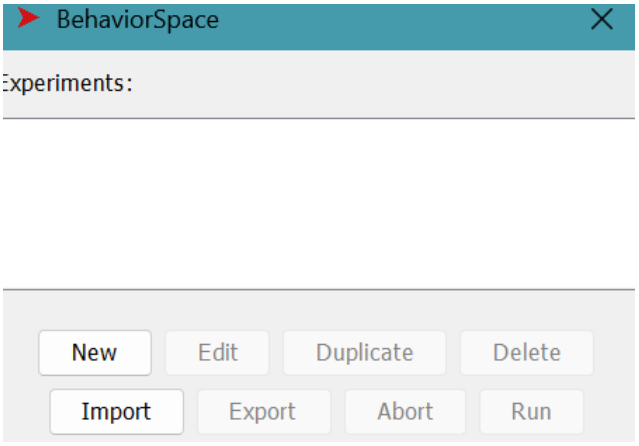
Preview Commands Editor

The **Preview Commands Editor** in NetLogo is a helpful tool that allows you to test your setup and go commands before running an experiment. It ensures that your commands are valid and that the model behaves as expected, helping to catch errors early and avoid wasting time on misconfigured experiments.

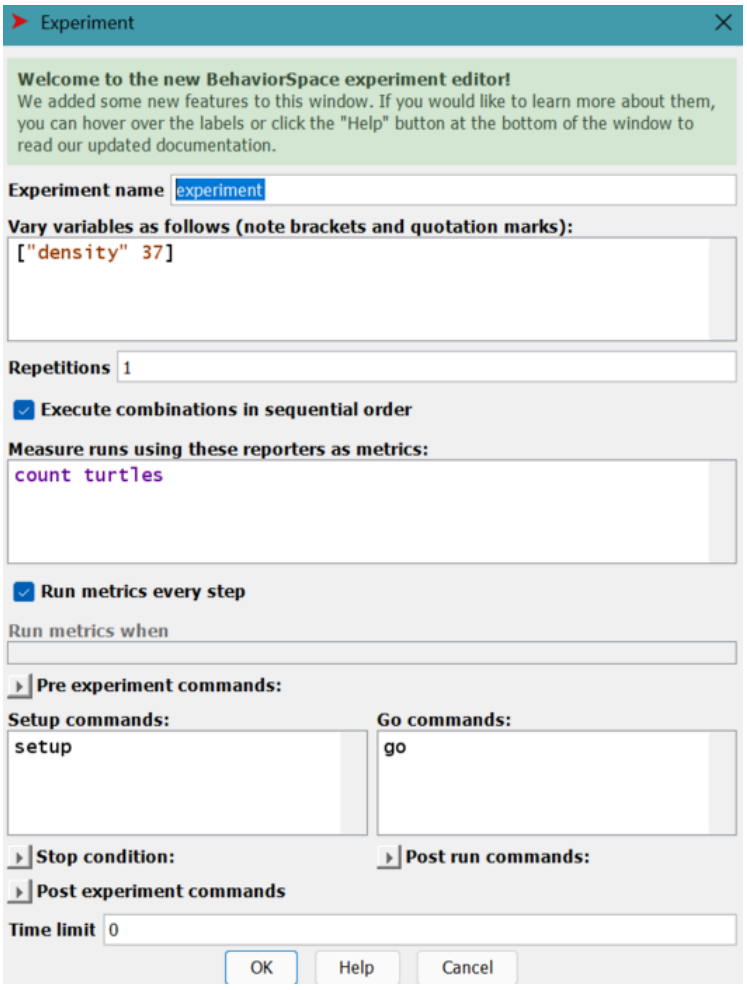


BehaviorSpace

BehaviorSpace in NetLogo is a tool that allows you to run and analyze parameterized experiments by automating the execution of your model with different variable combinations. The **Preview Commands Editor** is connected to **BehaviorSpace** because it allows you to validate and fine-tune the **setup** and **go** commands used in your experiment before running it.



Pressing the **New** button brings up the following window:



- **Experiment Name**
  - **Purpose:** Assigns a name to the experiment for identification.
- **Vary Variables as Follows**
  - **Purpose:** Specifies which variables will change during the experiment and their values.
  - **Example:** `["density" 30 40 50 60 70]`
- **Repetitions**
  - **Purpose:** Defines how many times the experiment will repeat for each parameter combination.
  - **Example:** Set it to `100` to average the results over 100 runs at each density value.
- **Execute Combinations in Sequential Order**
  - **Purpose:** Ensures that variable combinations are tested in the order specified.
- **Measure Runs Using These Reporters as Metrics**
  - **Purpose:** Specifies the values to record during the experiment.
  - **Example**: `count fires`
- **Run Metrics Every Step**
  - **Purpose:** Records the specified metrics at every step of the simulation.
- **Pre Experiment Commands**
  - **Purpose:** Commands to initialize the simulation before each run.
- **Stop Condition**
  - **Purpose:** Defines when the simulation should stop.
- **Time Limit**
  - **Purpose:** Specifies a maximum runtime for the simulation in seconds (0 means no limit).

Each experiment in NetLogo can be exported as an **.xml file**, allowing it to be reused or shared later to ensure reproducibility.

NetLogo also efficiently bundles all components—**interface elements (in XML)**, the **Info tab**, and the **Code tab**—into a single **.nlogo file** for easy management and distribution.

The Info Tab

NetLogo includes a built-in **documentation tool** that helps users effectively document their models. This tool's structure can also serve as a skeleton for documenting other projects. It typically follows the structure below:

- **What is it?**
  - A brief overview of the model and its purpose.
- **How it works?**
  - Explains the underlying mechanics, rules, and logic driving the model's behavior.
- **How to use it?**
  - Instructions for running the model, including details on controls, sliders, buttons, and other interface elements.
- **Things to Notice**
  - Key behaviors or patterns to observe when running the model.
- **Things to Try**
  - Suggestions for experimenting with the model, such as changing parameters or testing specific scenarios.
- **Extending the Model**
  - Ideas for adding new features or expanding the model's functionality.
- **NetLogo Features**
  - Highlights specific NetLogo commands or tools used in the model.
- **Related Models**
  - Lists similar models in the NetLogo library or other related projects.
- **Credits and References**
  - Acknowledgments for contributors and references to materials that inspired or informed the model.
- **How to Cite**
  - Citation format for the model, useful for academic or research purposes.
- **Copyright and License**
  - Details on the model's licensing terms and copyright information.

Utilizes Markdown syntax, just like this document.

Exercise: Create a reporter called `percent-burned` that calculates and reports the percentage of burned trees in the simulation.

Modify the code to use `neighbors` instead of `neighbors4`, allowing the fire to spread to all 8 neighboring patches.

- Set up a parameterized experiment in BehaviorSpace with the following conditions:
  - `density` set to 37%.
  - 100 repetitions of the experiment.
  - Metrics logged at every simulation step.
- Analyze the experiment results at step 25:
  - Calculate the mean and standard deviation of the `percent-burned` values across all runs.
  - Determine if any of the runs reached 90% burned trees.

Solution:
▶ **Click to show/hide solution**

Exercise: Update the model so that fires start from all edges of the grid lattice instead of just one side.

Run the simulation and observe how the fire spreads:

- Determine the `density` percentage at which the phase transition occurs when fire spreads from all directions.
- Compare the results to the original setup where fire started from one edge.
- Is the fire approximately four times as effective when starting from all edges? Explain your observations.

Solution:
▶ **Click to show/hide solution**

Exercise: Extend the functionality of the model by incorporating the effect of wind on fire spread.

This means that fire can spread not only to immediate neighbors but also to the neighbors of those neighbors.

- Modify the fire-spreading logic to include wind as a factor influencing the spread of fire.
- Determine the **phase transition** density for fire spread under wind influence.

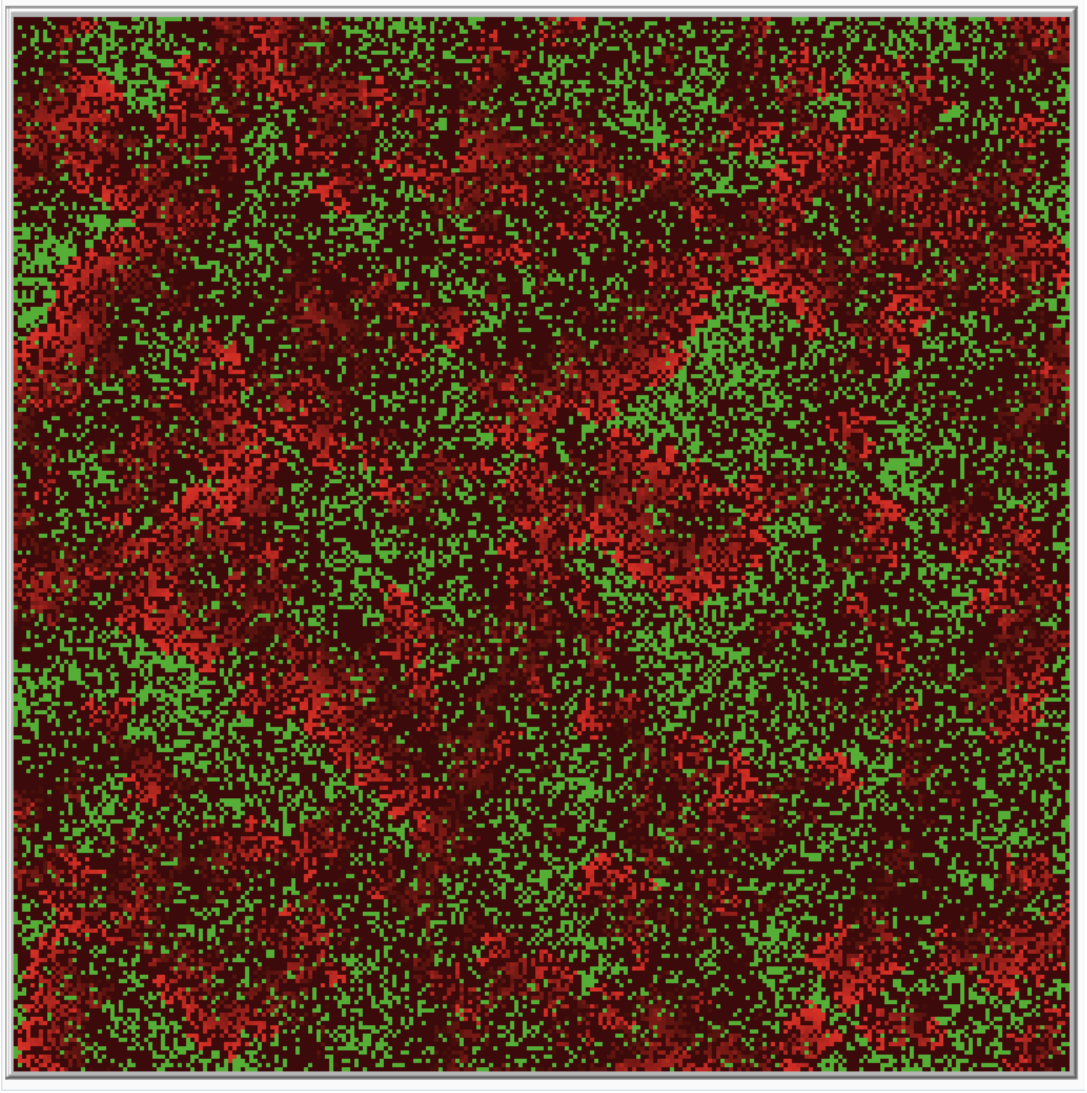
Solution:
▶ **Click to show/hide solution**

Exercise: Extend the functionality of the model by adding tree regrowth at each step.

Modify the code so that trees can regrow with a certain percentage per step, controlled by a **slider** (e.g., `regrowth-rate`).

- Add a slider to the interface to control the regrowth percentage.
- Update the `go` procedure to allow patches (previously burned or empty) to regrow into trees based on the slider value.
- Run your experiments again and observe how regrowth affects fire spread dynamics.
- Determine the new phase transition density with tree regrowth enabled.

Solution:
▶ **Click to show/hide solution**



Exercise: Update the model to calculate the percentage burned based on the number of burned patches currently on the map, rather than using the initial number of trees (since trees can now regrow).

- Modify the `percent-burned` reporter to dynamically calculate the percentage of burned patches at any given moment.
- Ensure the calculation accounts only for patches with `pcolor = black`.

Solution:

▶ **Click to show/hide solution**

Extra Exercise: Create and analyze a parameterized simulation using the following steps:

1. Set up a simulation in BehaviorSpace with:
   - `density` values ranging from 25% to 30% (in increments of 1%).
   - `regrowth-rate` values ranging from 2% to 5% (in increments of 1%).
   - 10 repetitions for each combination of parameters.
   - A time limit of 10 seconds per simulation.
2. Use the newly created `percent-burned` reporter to record the percentage of burned trees at the last step of each simulation.
3. Export the simulation results to a CSV file.
4. In Python:
   - Calculate the mean of the `percent-burned` values at the last step for each parameter combination.
   - Plot the results in 4 separate histograms, one for each `regrowth-rate` value (2%, 3%, 4%, and 5%).
   - Each histogram should display the percentage burned for `density` values from 25% to 30% (6 bars).

Schelling's Segregation Model

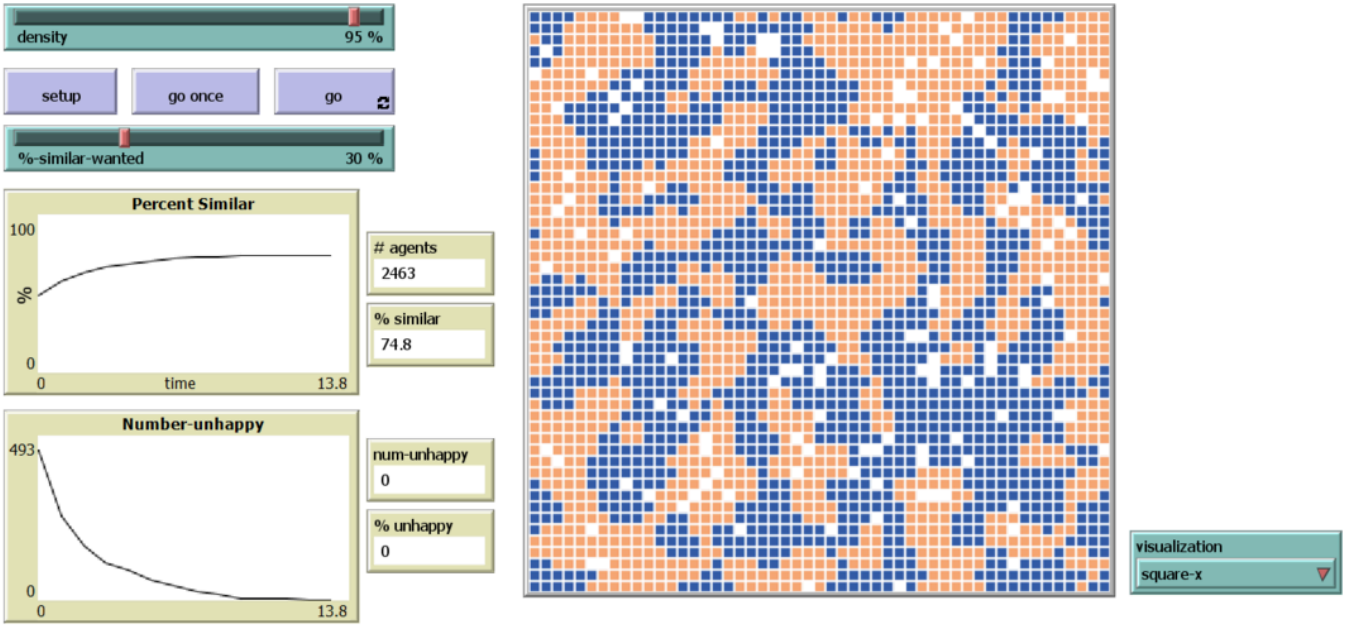
File > Models Library > Sample Models > Social Science > Segregation

Schelling's model of segregation has been regarded as one of the first agent-based models to address a significant social issue. It was created by Thomas Schelling, an economist who won the Nobel Prize for his contributions to economics during the Cold War. The segregation model naturally focuses on the social issue of segregation. He published it in 1972 and was originally called **Schelling's Tipping Model**.

In Schelling's original concept, the model represented a traditional **American urban landscape**, primarily inhabited by Black and White Americans. Racism and its resulting segregation were significant societal issues, and Schelling aimed to understand the mechanisms behind how racism influenced the urban landscape.

The basic idea was to create two types of agents, each with a preference for the composition of their neighboring agents expressed as a percentage. For example, a `similar-wanted` value of 30% means an agent will be considered `happy` if at least 30% of its neighbors are of the same type. This concept of happiness is crucial, as the goal of the model was to explore the tolerance level at which segregation does not occur while ensuring that all agents remain happy.

In this model, if an agent is unhappy, it attempts to move to a random empty location.



The `density` `setup` `go` and `go once` interface elements should be familiar by now. What they do here is exactly the same as what they did at the **Fire Model**.

There are four monitors with different reporters in the model:

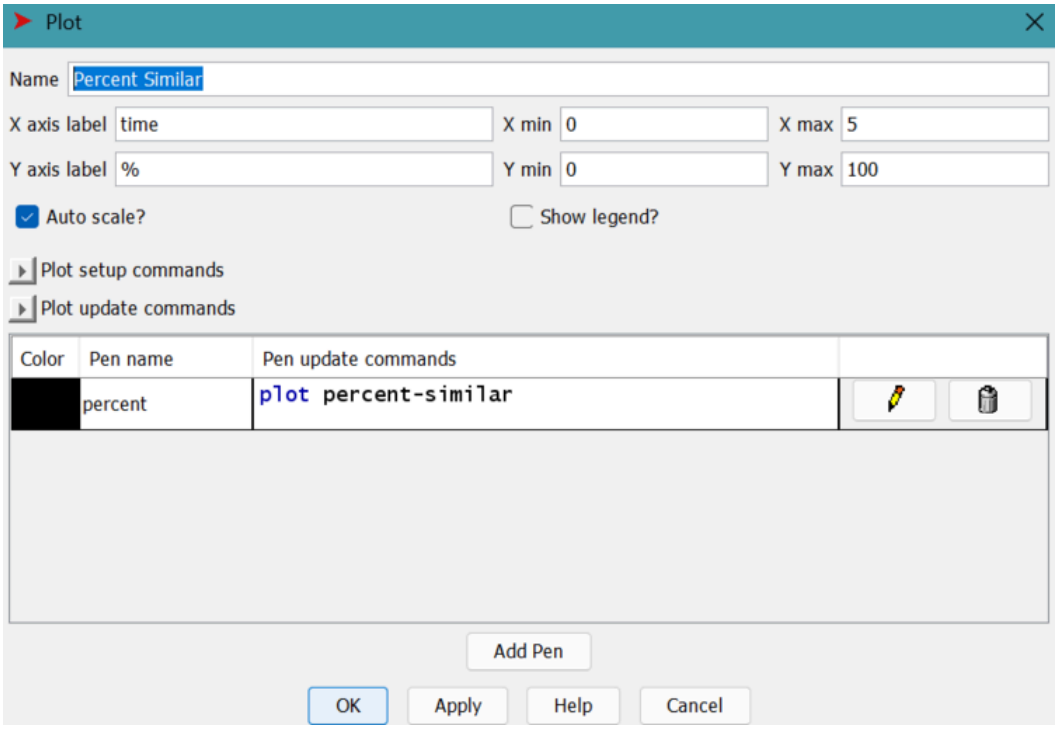
- One reports the **total number of agents**, which remains constant and reflects the `density` value.
- The `percent-similar` monitor calculates the average percentage of an agent's neighbors that are the same color as the agent.
- The `num-unhappy` monitor reports the total number of unhappy agents.
- The `percent-unhappy` monitor reports the percentage of unhappy agents on the grid.

Fun Facts:

1. In 1972, Schelling did not have access to computers capable of performing complex calculations in hours. Instead, he used a checkerboard with pennies and dimes, manually moving them around to observe the effects of his model.
2. Schelling described segregation as a "macro behavior" resulting from "micro motives." He later wrote a book titled *Micromotives and Macrobehavior*.

Plot Interface Element

The plot interface element in NetLogo allows us to visualize how a global variable changes over time, providing insights into how the system responds to small changes in the environment.



- **Plot Name**: Specifies the name of the plot, which will be displayed in the interface.
- **X-Axis Settings**:
  - Allows you to label the x-axis and set its minimum and maximum values.
- **Y-Axis Settings**:
  - Allows you to label the y-axis and set its minimum and maximum values.
- **Auto-Scale**:
  - Automatically adjusts the minimum and maximum values for better scaling during the simulation.
- **Legend**:
  - Adds a legend to identify what each pen represents.
- **Setup Commands**:
  - You can add commands to run before plotting begins, such as initializing variables or preparing the environment.
- **Pen Options**:
  - **Multiple Pens**: Track multiple variables on the same plot using different pens with different colors.
  - **Pen Names**: You can assign a unique name to each pen for clarity.
  - **Plot Types**: Pens support bar plots, point plots, and line plots. In your example, a line plot is used.
  - **Plotting Intervals**: You can specify how frequently the pen updates, controlling the time intervals for plotting.
  - **Basic Pen Updates**: A typical pen update looks like the following: `plot <reporter>`

Exercise: Explore the tipping points in the model with a density of **95%**.

- Find the tolerance level where the number of unhappy agents reaches 0, and no segregation occurs.
- Identify the tolerance level where more than 90% of agents remain unhappy, and the simulation does not converge (does not end).

While studying population dynamics of two groups of equal size, Schelling found a threshold (B_{seg}) such that:

- $(B_{\text{a}} < B_{\text{seg}})$ leads to a random population configuration.
- $(B_{\text{a}} \geq B_{\text{seg}})$ leads to a segregated population.

The value of (B_{seg}) was approximately $(\frac{1}{3})$.

The Code

So what do the `X` marks on the grid represent when `setup` is run?

The `X` marks indicate agents who are currently unhappy with their living situation. These agents will move to a new location in the next simulation step. Agents that are not marked with `X` are satisfied and will remain in their current position.

Global Variable Definitions

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

Turtle Properties (each turtle will have these)

```
turtles-own [
  happy?           ; 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?
  other-nearby     ; how many have a turtle of another color?
  total-nearby     ; sum of previous two variables
]
```

Setup

Breeds are **not used in this model** because agents are only created during the setup phase, their color (representing race) does not change, and their movement is not influenced by their color. This simplifies the implementation.

```
to setup
  clear-all
  ; create turtles on random patches.
  ask patches [
    set pcolor white
    if random 100 < density [   ; set the occupancy density
      sprout 1 [
        ; 105 is the color number for "blue"
        ; 27 is the color number for "orange"
        set color one-of [105 27]
        set size 1
      ]
    ]
  ]
  update-turtles
  update-globals
  reset-ticks
end
```

Helper Functions

The `[color] of myself` is used to explicitly reference the `color` of the turtle executing the command (the caller), distinguishing it from the `color` of the agents in the agentset (`turtles-on neighbors`).

For visualizations, there are two `if` statements controlling different representations. In the case of the square visualization, there is an `ifelse` within the `true` branch of the first `if`. If the agent is happy, it is displayed as a square; if not, it is displayed as an X shape.

```
to update-turtles
  ask turtles [
    ; in next two lines, we use "neighbors" to test the eight patches
    ; surrounding the current patch
    set similar-nearby count (turtles-on neighbors)  with [ color = [ color ] of myself ]
    set other-nearby count (turtles-on neighbors) with [ color != [ color ] of myself ]
    set total-nearby similar-nearby + other-nearby
    set happy? similar-nearby >= (%-similar-wanted * total-nearby / 100)
    ; add visualization here
    if visualization = "old" [ set shape "default" set size 1.3 ]
    if visualization = "square-x" [
      ifelse happy? [ set shape "square" ] [ set shape "X" ]
```

```
    ]
  ]
end
```

Here, two local variables are defined. These should not be confused with agent variables, as they are not updated here but are only used within the `sum` command. Their role is determined by the calling order in the `setup` command.

```
to update-globals
  let similar-neighbors sum [ similar-nearby ] of turtles
  let total-neighbors sum [ total-nearby ] of turtles
  set percent-similar (similar-neighbors / total-neighbors) * 100
  set percent-unhappy (count turtles with [ not happy? ]) / (count turtles) * 100
end
```

Main Loop

If all turtles are happy, stop the simulation; otherwise, move the unhappy turtles and update both agent and global variables.

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

```
to move-unhappy-turtles
  ask turtles with [ not happy? ]
    [ find-new-spot ]
end
```

The `find-new-spot` command is applied to all unhappy turtles. It rotates the agent in a random direction (right turn) and moves it forward by a random distance between 0 and 10 patches. If the destination is already occupied by another agent, the turtle continues searching for a new spot on the grid.

This computation might seem inefficient. Let's create a new command that identifies empty patches on the grid and moves agents randomly to one of those patches.

```
to find-new-spot
  rt random-float 360
  fd random-float 10
  if any? other turtles-here [ find-new-spot ] ; keep going until we find an unoccupied patch
  move-to patch-here  ; move to center of patch
end
```

Exercise: Create a new `find-new-spot` command to optimize agent movement.

- Instead of having the agent repetitively move to random spaces, first infer all empty patches on the grid where no turtles are present.
- Update the agent to move randomly to one of the inferred empty patches.
- How does this updated command perform compared to the original one?

Solution:
▶ **Click to show/hide solution**

Modifying the Code

Exercise: Add a **goodness** level to the agents in the simulation and modify their behavior based on this attribute.

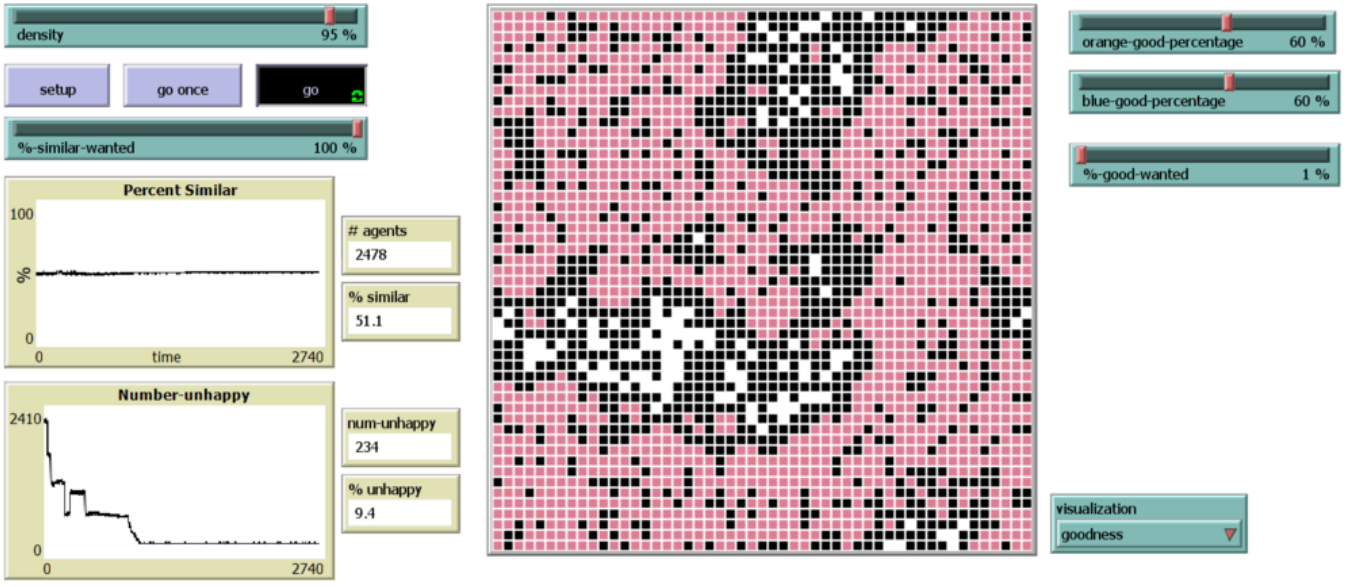
1. For the `setup` stage, add two sliders:
   - `orange-good-percentage`: Controls the percentage of orange agents who are "good" (e.g., 80% means 80% of orange agents are good).
   - `blue-good-percentage`: Controls the percentage of blue agents who are "good".
2. Create a slider called `%-good-wanted` to control the percentage of good neighbors required for an agent to be happy.
3. Modify the model dynamics so that:
   - Agents prioritize having enough "good" neighbors based on `%-good-wanted`.
   - If the required number of good neighbors is not met, agents fall back to checking similarity in race (`%-similar-wanted`).
   - If neither condition is met, the agent becomes unhappy and moves.
4. Run the simulation with the following parameters:
   - Set `orange-good-percentage` and `blue-good-percentage` to 50%.
   - Test what happens when `%-good-wanted` is larger than `%-similar-wanted`.
5. Observe and analyze the results:
   - How does the newly added "good" attribute affect segregation?
   - How does it influence tipping dynamics when agents move?

Solution:
▶ **Click to show/hide solution**

Exercise: Can you think of any other visualization types that could be used here? Instead of showing the two types of agents, one could display one of their attributes to provide a different perspective.

Add at least two new visualizations!

Solution:
▶ **Click to show/hide solution**



Extra Exercise: Modify the code to include a controllable number of agent types in the environment, rather than just orange and blue.

- Add a new slider to control the number of agent types.
- Ensure each agent type is assigned a unique color dynamically based on the number of types specified.

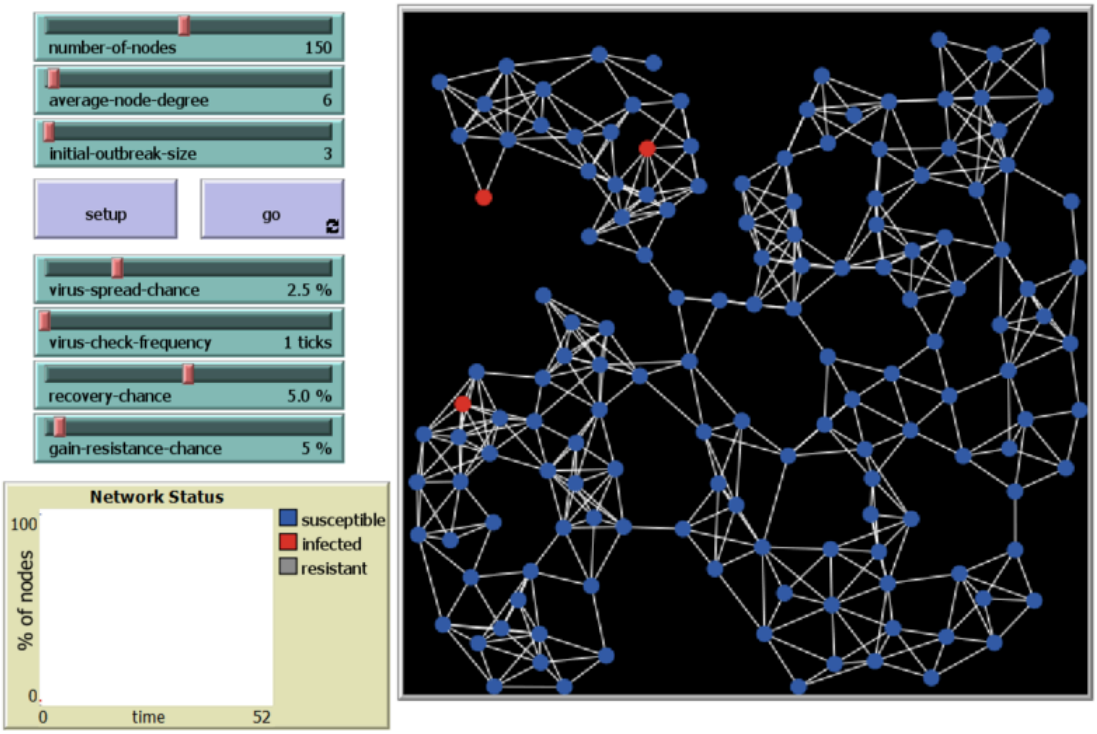
- Update the dynamics to account for similarity checks across all agent types.

Virus on a Network Model

File > Models Library > Sample Models > Networks > Virus on a Network

The **Virus on a Network** model was created by Uri Wilensky in 2008 to simulate the spread of a virus within a social network. The simulation begins with a network-building process, where parameters such as `number-of-nodes`, `average-node-degree`, and `initial-outbreak-size` define the structure of the network. This setup creates a network that closely resembles the structure of a social network. The model uses agents and **links** to represent nodes and their connections, respectively.

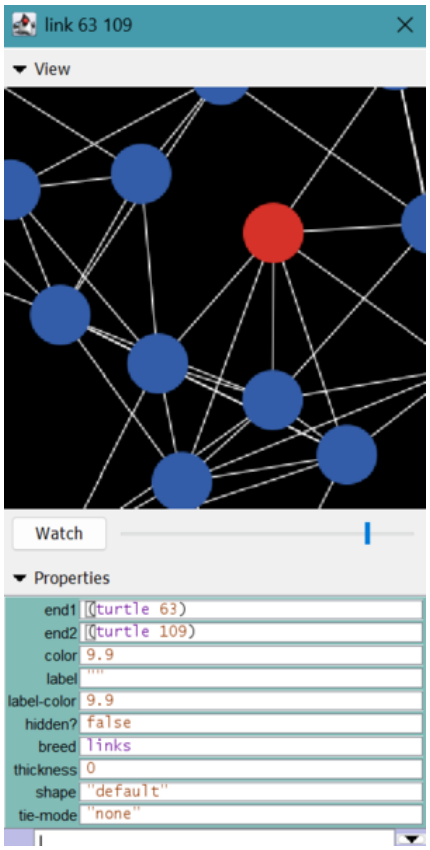
> **Remember:** Links are also considered as **agents** in the NetLogo environment.



This graph does not follow a Rényi or Barabási graph but instead represents a **random geometric graph (RGG)**. In graph theory, a **random geometric graph** is the simplest type of spatial network. It is an undirected graph created by randomly placing *N* nodes in a metric space and connecting two nodes with a link if and only if their distance is within a specified range, such as smaller than a given neighborhood radius, *r*.

Links

Links are used to represent connections or relationships between agents. They are commonly used in models involving networks, such as social networks, transportation systems, or ecological relationships. Link also have properties and can be inspected the same way as **turtles or patches**.



Links have the following **unique** properties compared to turtles and patches:

- `end1` **and** `end2`: Represent the two agents connected by the link, specifying the source and target nodes.
- `hidden?`: Indicates whether the link is hidden from view, similar to how it is used in the Fireflies simulation.
- `thickness`: Specifies the visual thickness of the link on the grid.
- `shape`: Defines the shape of the link. The default is a straight line, but other options are available (e.g., arrows for directed links).
- `tie-mode`: Tie connects two turtles so that the movement of one turtles affects the location and heading of another. When a link's `tie-mode` is set to "fixed" or "free" `end1` and `end2` are tied together. If the link is directed `end1` is the "root agent" and `end2` is the "leaf agent". That is when `end1` moves (using `fd`, `jump`, `setxy`, etc.) `end2` also moves the same distance and direction.

When a link between two nodes is **undirected**, `end1` is always the older node, meaning the node with the lowest `who` ID. However, when the link is **directed**, `end1` represents the source node, and `end2` represents the target node.

Examples:

Create an undirected link between `turtle 0` and `turtle 1`:

```
ask turtle 0 [ create-link-with turtle 1 ]
```

Create an directed link between `turtle 0` and `turtle 1`:

```
ask turtle 0 [ create-link-to turtle 1 ]
```

Query all links connected to `turtle 0`:

```
ask turtle 0 [ show link-neighbors ]
```

Change link properties (the same as with turtles or patches):

```
ask links [ set color red set thickness 0.5 ]
```

> **Note:** Once the first link has been created directed or undirected, all unbreeded links must match (links also support breeds, much like turtles).

Network Building

Let's see how it is done in the code:

```
turtles-own
[
  infected?           ;; if true, the turtle is infectious
  resistant?          ;; if true, the turtle can't be infected
  virus-check-timer   ;; number of ticks since this turtle's last virus-check
]

to setup
  clear-all
  setup-nodes
  setup-spatially-clustered-network
  ask n-of initial-outbreak-size turtles
    [ become-infected ]
  ask links [ set color white ]
  reset-ticks
end
```

The `setup` command, when called, first creates all the nodes in the network based on the `number-of-nodes` parameter. It then creates links between these nodes according to the RGG process. After that, it randomly selects `initial-outbreak-size` turtles to become infected. Finally, it sets the color of all links to white.

Node Setup:

```
to setup-nodes
  set-default-shape turtles "circle"
  create-turtles number-of-nodes
  [
    ; for visual reasons, we don't put any nodes *too* close to the edges
    setxy (random-xcor * 0.95) (random-ycor * 0.95)
    become-susceptible
    set virus-check-timer random virus-check-frequency
  ]
end
```

At the start of the simulations are agents becomes susceptible (not infected nor resistant to the virus). This also sets their color blue. Then their virus check timer becomes a number between 0 (inclusive) and `virus-check-frequency - 1` (exclusive)

```
to become-susceptible  ;; turtle procedure
  set infected? false
  set resistant? false
  set color blue
end
```

The `num-links` is the number of links needed to be created so that the network will have a proper `average-node-degree`. We know on average how many links will be connected to a node, the number of nodes, and also that are network is undirected, so the calculation is divided by 2.

While the global link count remains below the required threshold, an agent is selected from a filtered agentset. This agentset consists of all other turtles that are not part of the calling agent's link-neighborhood. From this set, the closest agent is chosen based on `distance myself`. If a valid choice is made (i.e., the selected agent is not empty), an undirected link is created between the two agents.

The second line is purely for visualization purposes, using spring and force-based physics to adjust the positions of the nodes.

```
to setup-spatially-clustered-network
  let num-links (average-node-degree * number-of-nodes) / 2
  while [count links < num-links ]
  [
    ask one-of turtles
    [
      let choice (min-one-of (other turtles with [not link-neighbor? myself])
```

```
                  [distance myself])
      if choice != nobody [ create-link-with choice ]
    ]
  ]
  ; make the network look a little prettier
  repeat 10
  [
    layout-spring turtles links 0.3 (world-width / (sqrt number-of-nodes)) 1
  ]
end
```

The Main Loop

If all turtles are either resistant or susceptible, the simulation stops. Otherwise, each agent increments its `virus-check-timer` by 1. When the timer reaches its threshold, it resets. The main loop concludes with the virus spreading (handled in a separate command) and the agents performing a virus check.

```
to go
  if all? turtles [not infected?]
    [ stop ]
  ask turtles
  [
     set virus-check-timer virus-check-timer + 1
     if virus-check-timer >= virus-check-frequency
       [ set virus-check-timer 0 ]
  ]
  spread-virus
  do-virus-checks
  tick
end
```

This command asks the infected agentset to check all their non-resistant neighbors and attempt to infect them based on the `virus-spread-chance`. If successful, the neighboring agents become infected, their color changes to red, and their `infected?` variable is set to `true`.

```
to become-resistant  ;; turtle procedure
  set infected? false
  set resistant? true
  set color gray
  ask my-links [ set color gray - 2 ]
end

to become-infected  ;; turtle procedure
  set infected? true
  set resistant? false
  set color red
end

to spread-virus
  ask turtles with [infected?]
    [ ask link-neighbors with [not resistant?]
        [ if random-float 100 < virus-spread-chance
            [ become-infected ] ] ]
end
```

The virus check asks all turtles that are infected and due for a check to attempt recovery. If recovery is successful, the agent has a chance to gain resistance based on the `gain-resistance-chance`. If successful, the agent becomes `resistant`; otherwise, it becomes `susceptible`. If recovery is unsuccessful, the agent remains `infected`.

```
to do-virus-checks
  ask turtles with [infected? and virus-check-timer = 0]
  [
    if random 100 < recovery-chance
    [
      ifelse random 100 < gain-resistance-chance
        [ become-resistant ]
```

```
        [ become-susceptible ]
    ]
  ]
end
```

Exercise: Experiment with the variables that influence the dynamics of the virus:

- Adjust the following hyperparameters in both directions:
  - `virus-spread-chance`
  - `virus-check-frequency`
  - `recovery-chance`
  - `gain-resistance-chance`
- Observe and record the effects of these changes on the simulation dynamics.
- Analyze which hyperparameter the model is most sensitive to and explain why this might be the case.

Exercise: Increase the `number-of-nodes` and the `average-node-degree` hyperparameter.

- Observe how the changes affect the dynamics of the virus spread.
- Compare the susceptible/resistant ratio to the previous configuration.

Exercise: Set the `gain-resistance-chance` to 0%.

- Observe whether the model can recover from the virus without agents gaining resistance.
- With `virus-spread-chance` set at 2.5%, identify the tipping point where the virus can no longer spread effectively.

Exercise: Real computer networks, where viruses often spread, typically do not rely on spatial proximity like the networks in this model. Instead, they often exhibit a "scale-free" link-degree distribution, similar to those created using the Preferential Attachment (Barabási) model.

- Experiment with alternative network structures, such as Rényi (random networks) and Barabási (scale-free networks).
- Observe how the behavior of the virus changes under these different network structures.
- Compare the virus dynamics to the spatial network in the original model and analyze the differences.

Rényi Solution:
▶ **Click to show/hide solution**

Barabási Solution:
▶ **Click to show/hide solution**

Extra Exercise: As an optional exercise, think about extending the network to include additional virus variants or the ability for the virus to mutate and evolve over time.

- Consider how multiple virus variants could interact within the network.
- Design a solution to handle multiple virus types and their evolution over time.
- Implement your solution.

Tamás Takács

January 22, 2025