Exercise 3 – Segregation model

Open NetLogo

From the menu select File>Models Library.

Then select Sample Models>Social Science>Segregation.

Click on the Info tab and read associated information. Click back on the Interface tab and try playing with the model to see what it does. This is an implementation of a famous model produced by Thomas Schelling in the early 1970’s[1].

We will now examine the program code in detail.

Select the Code tab. You should see something similar to:

```
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?
]

turtles-own [
  happy?           ;; for each turtle, indicates whether at least %-similar-wanted percent
                   ;;  of that turtles' 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
]

to setup
  clear-all
  ;; create turtles on random patches.
  ask patches [
    if random 100 < density [   ;; set the occupancy density
      sprout 1 [
        set color one-of [red green]
      ]
    ]
  ]
  update-variables
  reset-ticks
end

;; run the model for one tick
to go
  if all? turtles [ happy? ] [ stop ]
  move-unhappy-turtles
  update-variables
  tick
end

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

;; move until we find an unoccupied spot
to find-new-spot
  rt random-float 360
  fd random-float 10
```

[1] Schelling, Thomas C. (1971) Dynamic Models of Segregation. Journal of Mathematical Sociology 1:143-186. http://www.stat.berkeley.edu/~aldous/157/Papers/Schelling_Seg_Models.pdf

```
  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

to update-variables
  update-turtles
  update-globals
end

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" ]
    if visualization = "square-x" [
      ifelse happy? [ set shape "square" ] [ set shape "square-x" ]
    ]
  ]
end

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

; Copyright 1997 Uri Wilensky.
; See Info tab for full copyright and license.
```

Discussion of the code:

Declarations:

Starting from the top of the code we can see two global variables defined using the "globals" command: percent-similar and percent-unhappy. Note that anything following a ";" is a comment.

Then four additional fields are added to turtles using the "turtles-own" command. Note that the question mark in "happy?" has no special significance it is just part of the variable name. However, it is often used as a convention when values will take a true / false value.

Next are defined the setup and go procedures.

Setup and go procedures:

Setup creates new turtles and ensures there is only a maximum of one agent in each patch. It does this by asking each patch to "sprout" an single turtle with probability proportional to the density value. Density is defined by a slider on the interface screen. Newly sprouted turtles execute a command to set their colour to either red or green (the "one-of" reporter returns a random selection from the list that follows it). Setup then calls another procedure called "update-variables" which we discuss later.

Go firstly checks if all turtles have their "happy?" value set to true. If so the program terminates using the "stop" command. Next the procedure "move-unhappy-turtles" is called followed by the "update-variables" procedure.

Move-unhappy-turtles procedure:

This procedure asks each turtle that is not happy (i.e. has its “happy?” variable set to false) to execute the procedure find-new-spot. The find-new-spot procedure moves the turtle in a random direction for a random distance until it finds an empty patch. It does this recursively. Note that the “move-to patch-here” command at the end ensures the turtle is placed in the exact centre of the patch before find-new-spot terminates.

Update-variables procedure:

This procedure calls two other update procedures. The first, update-turtles, updates the turtle variables and the second, “update-globals” updates the global variables.

Update-turtles procedure:

The update-turtles procedure determines if each turtle is happy or not and assigns a true / false value to the “happy?” turtle variable. It also updates the similar-nearby, other-nearby and total-nearby turtle variables. It calculates these by asking each turtle to check it’s neighbours (that is turtles on eight adjacent patches). The colours of the neighbours are compared to its own to determine if at least “%-similar-wanted” (a global variable set in a slider on the Interface screen) of them are the same colour. Consider the line:

```
set similar-nearby count (turtles-on neighbors) with [color = [color] of myself]
```

This sets the value of the (current) turtle variable “similar-nearby” to a count of a set of turtles on neighbouring patches (to the current turtle) that satisfy the condition after the “with”. In the condition “color = [color] of myself” the term “color” means the neighbours’ colour. The term “[color] of myself” means the color of the current turtle. The “myself” term is a special term that means the current (calling) turtle.

Consider the line:

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

This line applies the logic of the Schelling segregation rule. It sets the “happy?” turtle variable to true if “similar-nearby” (neighbours of the same colour) equals or exceeds the required threshold of “%-similar-wanted” (a global variable set on an Interface screen slider expressed as a percentage of all neighbours).

The final part of the procedure details with how the turtles are visualised in the environment dependent on the “visualization” global variable (set on an Interface screen dropdown selector).

Update-globals procedure:

The first two lines of the update-globals procedure create two new local variables (local to the procedure) using the “let” command. These store the sum of the turtle variables similar-nearby and total-nearby. From these the global variables percent-similar and percent-unhappy are calculated which store the overall state of the population and are used for display purposes on the Interface screen.

Task: Modify the model so that instead of just two colours agents take one of four colours (red, green, blue or yellow). Does this make any difference to the results of the model?

Task: Add a “Chooser” dropdown to the Interface screen (like the existing visualization one) that lets the user select between either two colours or four colours. Modify the program so that it uses the number of colours selected by the user.

Task: Modify the find-new-spot procedure so that instead of recursively moving in random directions to find an empty patch it moves to a random empty patch selected from all currently empty patches. Hint: this can be done in one line by asking patches.

Task: Create a button called “find-clusters” that when clicked counts number of connected clusters of the same colour. Display the result in a monitor on the interface. Two turtles are part of a connected cluster if they are the same colour and there is a route that connects them by only passing through turtles of the same colour as themselves. The route can be any length. Hint: this can be done with a recursive procedure.