Chapter 11 Exercise Hints and Solutions

Agent-based and Individual-Based Modeling: *A Practical Introduction, 2nd Edition*

Exercise 1

Part of a correct solution is creating a variable name (e.g., `let blue-patches`) that describes what the variable holds. Another part of the exercise is to show that the first six statements work correctly, which often takes considerable programming effort. We show examples of how to test statements, though many alternative ways could also work.

Solutions:

- `let green-turtles turtles with [color = green]`
  This statement can be checked via a test program that creates some turtles (which have random colors by default), moves them to random locations, then identifies the green ones and makes them bigger:

```
to test
  ca
  crt 100
  [ move-to one-of patches ]

  let green-turtles turtles with [color = green]
  ask green-turtles [ set size 5 ]

end
```

  Then one can look in the View for any discrepancies. If students do this, they will find many turtles that look green but are not big. They should (a) use Agent Monitors to see the color value of these turtles, and then (b) look in the Programming Guide to see whether those color values are NetLogo's "green" color. Colors 65 ("lime") and 75 ("turquoise") also can look green.

  (One potential error students could make is using `pcolor` instead of `color`, in which case no turtles will be big because `pcolor` is black for all patches.)

- `let greenish-turtles turtles with [color >= 50 and color < 60]`
  Consult the "colors" section of the Programming Guide to identify the color numbers for the shades of green. This statement can be tested the same way the previous one was, except that we need to create turtles with the full range of color shades. In the `crt` statement, set the turtle's color to a random number between 0 and 140.

- `let occupied-patches patches with [any? turtles-here]`
  (Using `any?` is preferable to the equivalent `count turtles-here > 0`.) This could be checked by entering these statements in the Command Center:
  `show min [count turtles-here] of occupied-patches` which should be greater than zero to verify that all occupied-patches have turtles on them, and

`show count turtles-on occupied-patches` which should equal the total number of turtles to verify that all turtles are on occupied patches.

- `let unoccupied-patches patches with [not any? turtles-here]`
  This could be checked by entering these statements in the Command Center: `show max [count turtles-here] of unoccupied-patches` which should be zero to verify that no unoccupied-patches have turtles on them, and `ask turtles-on unoccupied-patches [show "HEY!"]` which verifies that no turtles are on unoccupied patches.

- `let patches-with-two-turtles patches with [count turtles-here = 2]`
  One way to test this is: `ask patches-with-two-turtles [show count turtles-here]`.

- `let lower-right-turtles turtles with [pxcor >= (min-pxcor + (world-width / 2)) and pycor <= (min-pycor + (world-height / 2))]`
  (Other methods such as `with [pxcor > 0 and pycor > 0]` only work for some World configurations.) Remember that turtles can use their patches' variables. This code can be tested by asking `lower-right-turtles` to change their color, pcolor, size, etc., and examining the View.

- `let us-in-radius-3 turtles in-radius 3`

- `let others-in-radius-3 other turtles in-radius 3`

- `let all-my-out-links my-out-links` (Note that, at any one time, all links must be either directed or undirected; you cannot have some directed and some undirected unless they are of different breeds.)

- `let all-my-links my-links`

- `let the-turtles turtles-on (patch-set [neighbors] of out-link-neighbors)` (The trick here is reading the dictionary for `of` to realize that when `of` refers to an agentset such as `out-link-neighbors`, it produces a list. Then `patch-set` is required to turn that list into an agentset, because `turtles-on` requires an agentset as input, not a list. Another trick is that `[neighbors]`refers to neighbors of turtles, which is equivalent to neighbors of the patches the turtles are on.)

- `let turtles-in-radius-5 turtles in-radius 5`

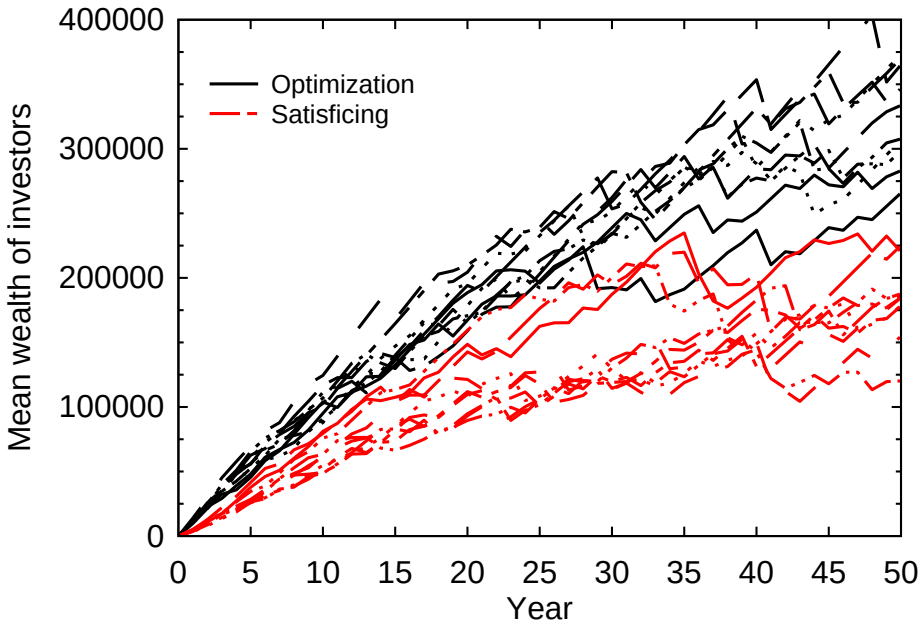
- `let turtles-on-patches-in-radius-5 turtles-on patches in-radius 5`

Exercise 2

Included with the instructor materials for Chapter 11 is an implementation of the model version described in Section 11.4 (`BusinessInvestors_Sect11-4_2ndEd.nlogo`). Part of revising the NetLogo program is changing the settings of the utility histogram plot so that it histograms the new version of utility in a useful way: values around 30,000 for "X max" and 1000 for the bar

interval work well. This histogram is useful for seeing how many investors have found a patch where their utility threshold is met.

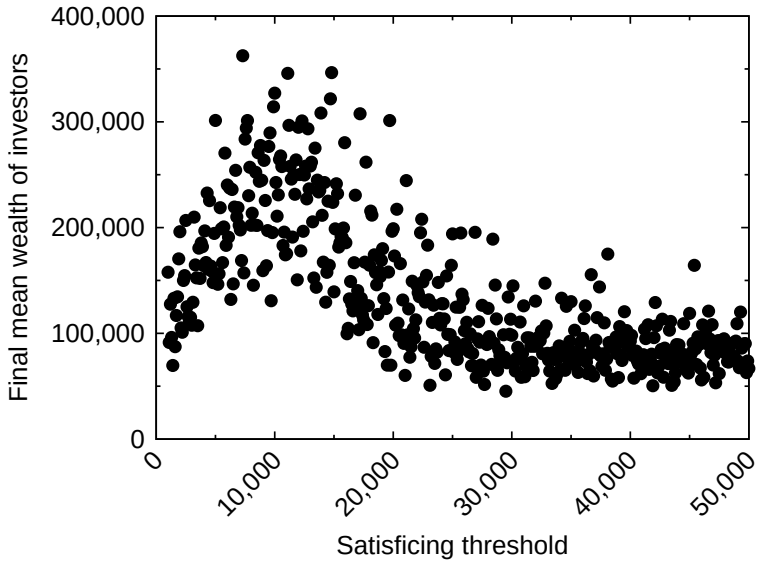
Students should produce a comparison of the satisficing vs. original version that looks something like the following graph. It is important that they illustrate the stochasticity of results in some way, by using multiple model runs; our graph below shows results of 10 runs of each version of the model as separate lines. Mean final wealth using satisficing should be about 2/3 that of the original optimization trait.

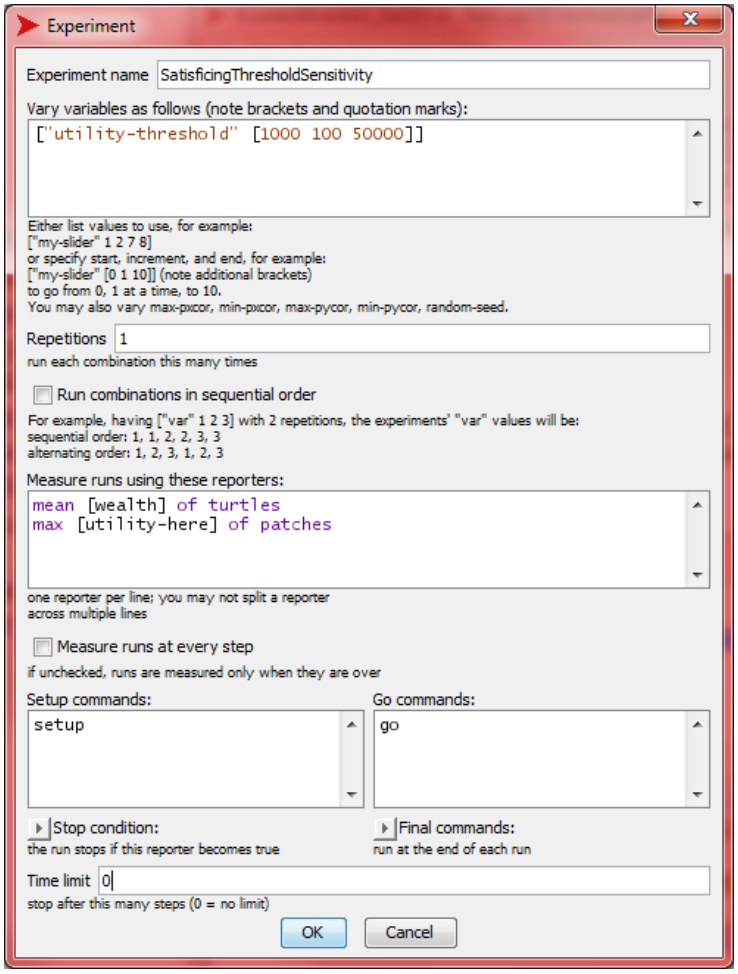


Exercise 3

Students should produce a sensitivity plot such as the following. We also show the BehaviorSpace setup used to produce this plot. Mean investor wealth goes up and then down, and finally plateaus, as the threshold goes up. The reasons for this response should be clear. As the threshold increases up to around 10,000, wealth increases because most investors can find still find patches where they meet the threshold as it increases. Above around 10,000, fewer and fewer investors can find a patch where the threshold is met; those that cannot keep moving randomly. With the threshold above around 25,000, few if any investors can meet their satisficing condition so they all just move randomly throughout the simulation.

To understand why results are so stochastic (noisy), it is useful to have BehaviorSpace write out the highest utility among all the patches, for each model run (`max [utility-here] of patches`). In the results plotted below, this maximum utility ranged from 16,650 to 60,700 in 491 model runs.



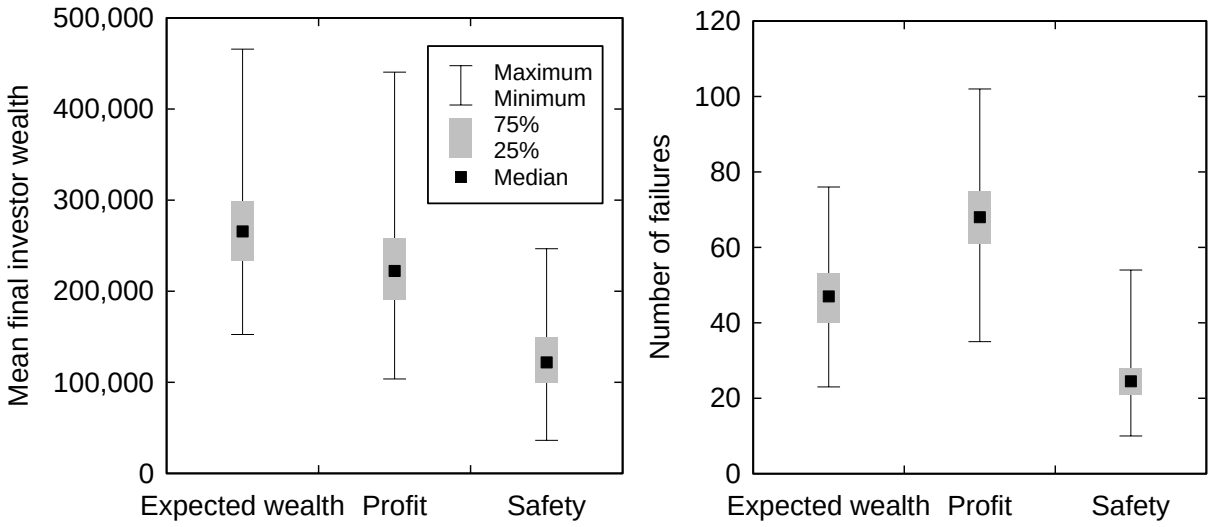


Exercise 4

An implementation of the Business Investor model of Section 11.5 is provided (`BusinessInvestors_Sect11-5_2ndEd.nlogo`). It includes all three of the alternative objective functions; choose a function by changing which statements are commented out in the reposition procedure. Be sure that students use the correct objective not just in the reposition procedure but also when they update the investors' `utility` state variable (in our example code, this is in `do-accounting`).

It is good to check to see whether the students have been updating the Info tab in their NetLogo files as they make these modifications (as well as testing their new objective functions).

In their comparison of the three objective functions, it is again important to include replicates to address the stochastic nature of this model. The following graphs show the distribution of results in 1000 runs of each version of the model.



We can also examine mean results over the 1000 runs. Compared to the baseline version with investors maximizing expected future wealth, the objective of maximizing profit regardless of risk actually reduces (by ~15%) mean investor wealth, while substantially increasing the number of failures—by about 45%. The objective of maximizing probability of staying in business reduces mean investor wealth by just over 50% but also reduces the number of failures by almost 50%.