Chapter 6 Exercise Hints and Solutions

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

Exercise 3

A NetLogo file (`Chapter6_Ex3_MemoryTest.nlogo`) implementing the test algorithm is provided with the instructor materials. The algorithm for remembering a path is simple but works surprisingly well. (Turn the speed slider down to watch the turtle move.)

However, the turtle does not retrace its path exactly, for two reasons.

First, on its very first move of the `go-back` procedure, the turtle faces the first patch on the memory list, which is the last patch it moved to, which is the patch it is still in. So the turtle faces the center of its current patch and moves forward one patch length, which causes it to move off in a random-seeming direction. (Remember that the turtle is moving in continuous space, not jumping from patch-center to patch-center.) This problem can be remedied by skipping the first patch on the memory list. Simple change `foreach path` to `foreach but-first path`.

The second reason the turtle does not exactly retrace its route is of course because the `go-back` procedure causes it to face the centers of the cells it was in, whereas the turtle did not travel from cell-center to cell-center.

How could you make turtles return to exactly where they have been? One alternative is for them to remember the exact location they were in instead of what patch they were in. The memory list could hold sub-lists that each contain the turtle's `xcor` and `ycor` values at the end of each move. A more clever way is to memorize the turtle's heading instead of its patch. Use this code in `setup`:

```
  crt 1
  [
    set color red  ; So we can tell initial path from return path
    set path (list heading)  ; Initialize the path as a list of HEADINGS

    pd

    ; Move randomly and memorize the HEADINGS at each step.
    ; Each new heading is put at the *start* of the list
    repeat 100
    [
      rt (random 91 - 45)
      fd 1
      set path fput heading path
    ]

  ]
```

and this in `go-back`:

```
    foreach but-last path   ; Executes once for each HEADING on list
```

```
; Set heading to the opposite of memorized one
[ a-heading -> set heading (a-heading + 180)
  fd 1
]
```

The file Chapter6_Ex3_MemoryTest_MemHeadings.nlogo implements this approach.

Exercise 4

The Butterfly Model, with the test output described in Section 6.3.9 and the mistake found in Section 6.3.10 corrected, is available in the instructor materials (`ButterflyModel_Ch6_Ex4.nlogo`). The results are analyzed (as in Figure 6.2) in the Excel spreadsheet `Ch06_Ex4_TestOutput.xlsb`. The example result obtained in this spreadsheet is that, with $q = 0.4$, butterflies moved to the highest neighbor on 48.1% of their moves. (Students should get values close to, but not exactly the same as this.)

The frequency of moving to the highest patch is predicted in Section 6.3.9 to be $q + (1-q)/8$. For $q = 0.4$, this predicted frequency is 0.475, or 47.5% of moves.

How can you decide whether the observed result (48.1%) is close enough to the prediction of 47.5%? One simple way is just to repeat the simulation to see how variable its results are. (They are somewhat variable, but always above 47.5%). Another way is to repeat the simulation using more turtles to get a higher sample size and see if the difference is smaller. (It is not.)

(One potential NetLogo problem in doing such experiments: NetLogo may refuse to delete the output file during `setup`. The reason is probably because the file is still open in your spreadsheet software.)

One cause for this discrepancy is the issue mentioned in Section 6.3.9: near the top of the artificial hill, several neighbor patches can have exactly the same, highest elevation. This makes it more likely that butterflies move to the highest neighbor. The butterflies spend many ticks near the hilltop, so this discrepancy can have strong effects. This issue can be dealt with using the trick introduced in Exercise 4 of Chapter 4: add some random noise to patch elevations so it is unlikely that multiple neighbors have exactly the same elevation.

Once this discrepancy is dealt with, the results of this test should be much closer to the predicted value.

Exercise 5

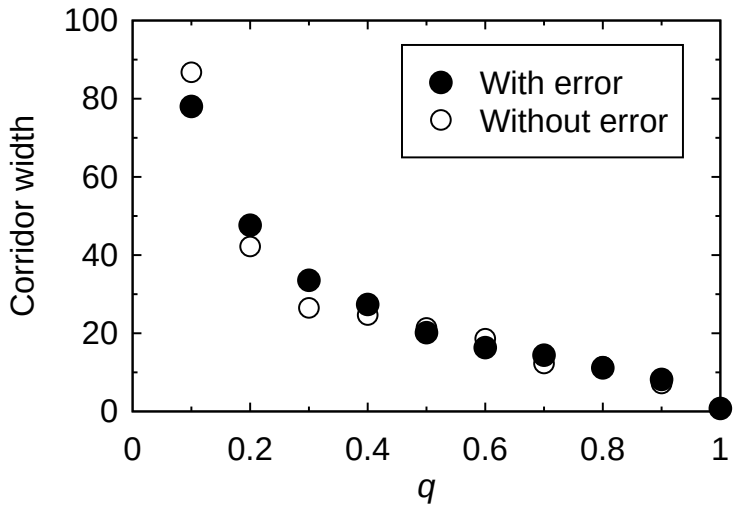
Both a version of the model (with the mistake) and an example analysis spreadsheet are provided as instructor materials (`ButterflyModel_Ch6_Ex5.nlogo` and `Ch06_Ex5_TestOutput.xlsb`).

Students should repeat this test after correcting the mistake, to confirm that the `uphill` problem was the only one.

Please be aware that the mistake caused by using `uphill` will not be apparent in an artificial landscape using an integer instead of the value "3.8" in the equation used to set elevations, e.g., `set elevation 200 + (100 * (sin (pxcor * 4.0) + sin (pycor * 4.0)))`. The error disappears when an integer is used because the hilltop has multiple patches with exactly the same, highest, elevation.

Exercise 6

The key results of the Chapter 5 analysis are those in figures 5.3 and 5.4. Students should reproduce these figures both before and after fixing the mistake identified in Section 6.3.10. It appears that the mistake has little consistent effect compared to the model's variability; see the example results below. The effect is probably small because the mistake only affects movement when the butterflies are near or at the top of a hill, in which case they are unlikely to increase the corridor-width measure much. (Because the effects of the error are small, replication is needed to evaluate them definitively; we introduce that technique in Chapter 8.)



Exercise 7

This exercise is very important. The Culture Dissemination model is completely new in the 2nd edition, and much simpler and more appropriate for beginners than the 1st edition's Marriage model. We simplified the exercise to keep beginners from being overwhelmed or, much worse, skipping the exercise because of its complexity. This exercise is important for reinforcing the importance of code testing, recognizing the kinds of mistakes that are very common in NetLogo, learning testing techniques, and for practicing the critical skill of writing and analyzing output files. Any students not challenged by this exercise should also try Exercise 8.

This makes a good team exercise during a lab session; allow as much as two hours. Encourage students to follow the steps in Section 6.5.2, starting with a review of the code before they start trying to run it. The code review should find many of the mistakes—if the students do not find mistakes during code review, make them try harder!

During the exercise, students are likely to encounter several kinds of file problem that are not errors in this model code but general issues in using output files. First, students may occasionally get a runtime error when they click `setup`, saying that NetLogo had an error while running `file-delete`. This is usually because the student has left one of the test output files open in another program (e.g., Excel). The `setup` procedure tries to delete this file so that it is overwritten at the start of each model run; when the file is open in another program it is locked from being deleted. Trying to run BehaviorSpace experiments on multiple processors with file output turned on will also cause a run-time error.

Some other file problems that can occur when students use the temporary test files:

- Some test output files can become extremely large, too large to open in Excel. Make sure students run the model long enough for a thorough test but stop the model before the test output is too large. Or they can use a text editor (WordPad, Notepad++) to delete enough lines to open the file in their analysis software.

- If the program stops ungraciously (e.g., due to a Java error instead of a NetLogo runtime error, which sometimes happens), it may leave a test file open, which causes a NetLogo error the next time the model is run. If NetLogo refuses to set up or go because a file is open, simply go to the Command Center on the Interface and enter `file-close-all`.

- This model may run very slowly with the test output files on, because of the time it takes the computer to open and close files constantly. A simple solution is to comment out any test output code not being used.

A version of the code with the errors fixed (`CultureDissemination_Fixed.nlogo`) is available with the instructor materials.

Here are the known mistakes in the Culture Dissemination model code (you may find more).

1. The global variable `a-site-changed?` is never initialized, causing a run-time error when it is used in the `go` procedure. One solution is to add `set a-site-changed? false` to `setup`. However, see the next item in this list.

2. In the `interact` (or `go`) procedure, we need a statement setting `a-site-change?` to `false` before the `interact` submodel is executed. Otherwise, the `go` procedure thinks that sites continue to change and never stops. The simplest solution is:

   ```
   to interact  ; a patch procedure
     set a-site-changed? false
   ```

3. The two plots on the Interface still contain the pen update command `plot count turtles`. For the mean similarity line plot, this causes two points to be plotted each tick, one a zero (because `count turtles` is zero). One clue is that the number of ticks on the

plot is double the actual number of ticks. For the histogram, the pen update command makes the X axis wrong.

4. In `setup`, the statement `set test-output-on? false` should be commented out. It will keep the test output files from being written.

5. In the `interact` procedure's statement `let the-rand-number random 1.0`, `random` should be `random-float` to get a number between zero and one. This error causes sites to become more like their neighbors much more often than they should, because `random 1.0` is always zero. The mistake should be obvious from the test output.

6. In `update-similarity` the statement `set mean-similarity (mean-similarity / 4)` should be `set mean-similarity (mean-similarity / count neighbors4)` to account for edge patches that have fewer than 4 neighbors. This mistake reveals itself via edge and corner patches always having mean-similarity < 1.0. The problem should be clear from the test output.

7. In Setup, there is a cut-and-paste error in setting up test output files:
```
if file-exists? "interact-test-output.csv"
  [
    file-delete "update-test-output.csv"
    file-open "update-test-output.csv"
```

8. Once the third problem above is fixed, the Similarity histogram does not display similarity values of 1.0, even though it should be clear from other output that similarity values of 1.0 are extremely common. The problem is that the histogram needs to have its `X max` parameter set to a value greater than 1.2. This is because (a) NetLogo's histogram puts values of 1.0 in the bin 1.0-1.2, and (b) the bin 1.0-1.2 does not appear unless `X max` has a value greater than 1.2. (A value of 1.21 for `X max` works.)

Exercise 8

This alternative code testing exercise uses a model more complex than that of Exercise 7, and the mistakes are harder to find and fix. They are also much more representative of the mistakes and problems we encounter with every real model. One realistic aspect of this exercise is that some of the problems are in the model description as well as in the code: fixing some of the software problems requires fixing ambiguities and omissions in the ODD model description. While this exercise is quite difficult, it has the benefit of exposing students to more of the kinds of mistakes that they are very likely to encounter in their own work. This exercise should therefore be suitable for students with prior programming experience, or as a refresher in software testing for students that have completed more of the book's Part II.

As with the Culture Dissemination model, students should find many of the mistakes in a careful review of the code before starting anything else.

The output file issues described above for Exercise 7 are also (and especially) important for this exercise. In addition, the test output files are turned on and off via commenting (turning the

output statements into comments using ";"). To produce these files, students must carefully search for all the output statements and un-comment them.

Here are the known mistakes in the Harvester model software. A version of the code with these errors fixed and an updated ODD on the Info tab (`Harvesters_Fixed.nlogo`) is available with the instructor materials.

1. When the model is first run, an error message pops up saying that a patch's value of `resource` was negative, in the `grow-resource` procedure. (Note the good example of defensing programming, which also opens an Inspector to the patch. The Inspector does not appear until you switch back to the Interface.)

   This error should lead students to turn on the optional test output in that procedure, by un-commenting the file-related statements. It should be immediately clear from this output (if not already from the View) that the initial value of `resource` is zero in all patches. `Setup` is missing a statement to initialize `resource` to half of `max-resource`; add the statement to the existing `ask patches` statement that sets `pcolor`. Now patches should be a variety of green shades upon setup.

   Students should also use this test output file to verify that resource growth is calculated correctly, by coding the growth equation independently in a spreadsheet.

2. Now the model runs for a while but the same error eventually appears. It should be clear from the resource growth submodel that negative resource can only occur if the amount harvested exceeds the value of `resource`. This error reveals a fundamental problem with the model: the model description does not consider the possibility that harvesters take more resource than is available on their patch, while it is clear that selfish harvesters could do so. At this point students need to fix both the model description and the code. A simple approach is to add a rule to the harvest submodel limiting the harvester's value of `harvest` to no more than the resource available in the patch. At the end of this submodel, just before the `set energy` statement, add: `if harvest > resource [ set harvest resource ]`.

3. Now the students will be disappointed to find that the same error still appears. They should now test whether harvesters are harvesting the correct amount, by turning on the test output in the `harvest` procedure. Now that a test has been added to make sure `harvest` does not exceed `resource`, the test output statements need to be modified to include `resource`. Re-implementing the harvest procedure should show that there is no error in it and that harvesters never take more resource than is available in their patch.

   If the harvesters do not take more than what is available, yet patches still calculate a negative value of resource, what could be wrong? It may help to modify the error statement in `grow-resources` so it gives more information:

   ```
   error (word "Resource is negative with prev-resource: " prev-resource "
   harvest: " harvest-here-last-tick  " resource-growth: " resource-growth)
   ```

   This error statement will confirm that `prev-resource` is negative, even though `resource` could not have been negative on the previous time step and the harvesters are not allowed to

consume more resource than is available. You can give students a hint by telling them to look carefully at the model's schedule. One problem is that `grow-resources` calculates the previous time step's harvest by summing the harvest of the harvesters on that patch, but which harvesters are on which patch changes due to death, resettlement, and reproduction. Another problem is that, while we have told each harvester not to harvest more than the value of `resource`, we have not prevented multiple harvesters in the same patch from each harvesting the entire value of `resource`.

Solving this problem again requires a subtle but potentially important new assumption that must be added to the model description. A simple solution is to have each harvester deplete the resource as they harvest: add this to the end of `harvest-resources`: `set resource resource – harvest`, and then remove the subtraction of harvest at the beginning of `grow-resources`. But it is important to understand that this solution makes the order in which harvesters execute this method affect the results: the first selfish harvester gets first chance to use up the resource. Other solutions could also be defined and programmed; e.g., when the total harvest exceeds the patch's resource, then all harvesters have their harvest reduced by an equal percent to make up the deficit (or even to keep resource from falling below some threshold).

4. The "Harvesters" plot of the number of cooperative and selfish harvesters indicates that their numbers only go down, even though the View indicates that the number of selfish harvesters is increasing. That there is a problem can be confirmed by entering `show count turtles` in the Command Center. The cause of this problem is a common mistake that can be hard for beginners to find. Students should look at the plot code, which uses two "convenience" agentsets, `cooperative-harvesters` and `selfish-harvesters`. These agentsets are initialized in `setup` to hold the cooperative and selfish harvesters. The error is that these agentsets are not updated as harvesters die and reproduce, so they only hold the initial harvesters, which gradually die out even if the population grows. (The error also affects the output file.)

   The easiest solution is to re-build these two agentsets after the `maybe-die` and `reproduce` actions are completed. Add these lines to the `go` procedure, between `reproduce` and `update-outputs`:

```
set cooperative-harvesters turtles with [behavior-type = "cooperative"]
set selfish-harvesters turtles with [behavior-type = "selfish"]
```

5. Attempting to independently re-implement the Resettlement submodel (if not just reading the code) should quickly make it clear that the code completely ignores selfish harvesters that are dissatisfied with their harvest. The `go` procedure only executes `resettle` if harvest is less than the maintenance energy, which is the test for whether cooperative harvesters are dissatisfied. The best solution is probably to have all harvesters execute `resettle` and put separate tests for the two kinds of harvesters in that procedure.

6. Inspection of the main output file will find that the cumulative total harvest by selfish harvesters always nearly equals the cumulative harvest by cooperative harvesters, even though the results reported for each time step are different for the two harvester types. The mistake is a cut-and-paste typographical error in the statement that updates `cumulative-`

`harvest-selfish` in the `harvest-resource` procedure. Students should make sure that results in the output files are consistent with each other wherever they can be checked.

7. Careful inspection of the main output file will also find that the cumulative total harvest, for both types of harvesters, does not equal the sum of the daily harvest values. This is because the daily totals were calculated in the `update-outputs` procedure, as `sum [harvest] of cooperative-harvesters`. This approach misses the harvest by harvesters that died in the current time step. It is also incorrect because reproduction creates new harvesters that inherit the harvest variable values of their parents. The simplest solutions are (a) initialize harvest variables of new harvesters to zero, just as good practice, and (b) delete this daily output of total harvest and instead use the cumulative harvest values that are updated in the `harvest` procedure.

8. Make sure students try all four landscape types. When they try the normally distributed landscape, they will again get the error statement that resource is negative. They should immediately see from the Inspector that the value of `max-resource` is negative. This is of course because the normal distribution can produce negative values (try alternative values of `max-resource-sd` to see which are unlikely to produce any negative values). To solve this, students again need to update the model description and code with their assumption of how to handle negative values. A simple approach is just to set `max-resource` to a small number whenever a negative value is drawn (setting it to zero causes a division-by-zero error in `grow-resource`):

```
ifelse landscape-type = "normal"
[ ask patches
  [
    set max-resource random-normal max-resource-mean max-resource-sd
    if max-resource < 0.0 [ set max-resource 1.0 ]
  ] ]
```

9. In the procedure `to resettle`, this statement:
   `if (random-float 1.0 > dissatisfaction-resettle-prob)`
   should be:
   `if (random-float 1.0 < dissatisfaction-resettle-prob)`
   This is an example of an error that has no effect under the model's current parameterization (because the value of `dissatisfaction-resettle-prob` is 0.5), but would have strong effects with a different parameter value.