Exercise 6 – El Farol Bar model

Open NetLogo

From the menu select File>Models Library.

Then select Sample Models>Social Science>El Farol

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 by Brian Arthur (1994)[4]. He wanted to show how agents employing inductive reasoning (rather than ideal rationality) could be modelled in economic-like situations. Such agents are often termed “boundedly rational” because their have limits on their cognitive capacities.

Each agent (turtle) in the model stores a number of “predictor” rules (called strategies) that it uses to predict the future based on the past. It selects the best rule, based on how well it would have done in the past, to decide what to do. It constantly updates its evaluation of its rules based on the history of what all the other agents do.

The implementation is useful to understand since it makes extensive use of lists.

We will now examine the program code in detail.

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

```
globals [
  attendance        ;; the current attendance at the bar
  history           ;; list of past values of attendance
  home-patches      ;; agentset of green patches representing the residential area
  bar-patches       ;; agentset of blue patches representing the bar area
  crowded-patch     ;; patch where we show the "CROWDED" label
]

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
]

to setup
  clear-all
  set-default-shape turtles "person"

  ;; create the 'homes'
  set home-patches patches with [pycor < 0 or (pxcor <  0 and pycor >= 0)]
  ask home-patches [ set pcolor green ]

  ;; create the 'bar'
  set bar-patches patches with [pxcor > 0 and pycor > 0]
  ask bar-patches [ set pcolor blue ]

  ;; initialize the previous attendance randomly so the agents have a history
  ;; to work with from the start
  set history n-values (memory-size * 2) [random 100]
  ;; the history is twice the memory, because we need at least a memory worth of history
  ;; for each point in memory to test how well the strategies would have worked

  set attendance first history

  ;; use one of the patch labels to visually indicate whether or not the
```

[4] W. Brian Arthur (1994) “Inductive Reasoning and Bounded Rationality”, American Economic Review, 84,406–411. http://tuvalu.santafe.edu/~wbarthur/Papers/El_Farol.pdf

```
  ;; bar is "crowded"
  ask patch (0.75 * max-pxcor) (0.5 * max-pycor) [
    set crowded-patch self
    set plabel-color red
  ]

  ;; create the agents and give them random strategies
  ;; these are the only strategies these agents will ever have though they
  ;; can change which of this "bag of strategies" they use every tick
  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
    update-strategies
  ]
  ;; start the clock
  reset-ticks
end

to go
  ;; update the global variables
  ask crowded-patch [ set plabel "" ]
  ;; each agent predicts attendance at the bar and decides whether or not to go
  ask turtles [
    set prediction predict-attendance best-strategy sublist history 0 memory-size
    set attend? (prediction <= overcrowding-threshold)  ;; true or false
  ]
  ;; depending on their decision, the agents go to the bar or stay at home
  ask turtles [
    ifelse attend?
      [ move-to-empty-one-of bar-patches
        set attendance attendance + 1 ]
      [ move-to-empty-one-of home-patches ]
  ]

  ;; if the bar is crowded indicate that in the view
  set attendance count turtles-on bar-patches
  if attendance > overcrowding-threshold [
    ask crowded-patch [ set plabel "CROWDED" ]
  ]
  ;; update the attendance history
  ;; remove oldest attendance and prepend latest attendance
  set history fput attendance but-last history
  ;; the agents decide what the new best strategy is
  ask turtles [ update-strategies ]
  ;; advance the clock
  tick
end

;; determines strategy that would have predicted best results had it been used this round.
;; the best strategy is the one that has the sum of smallest differences between the
;; current attendance and the predicted attendance for each of the preceding
;; weeks (going back MEMORY-SIZE weeks)
;; this does not change the strategies at all, but it does (potentially) change the one
;; currently being used and updates the performance of all strategies
to update-strategies
  ;; initialize best-score to a maximum, which is the lowest possible score
  let best-score memory-size * 100 + 1
  foreach strategies [
    let score 0
    let week 1
    repeat memory-size [
      set prediction predict-attendance ? sublist history week (week + memory-size)
      set score score + abs (item (week - 1) history - prediction)
      set week week + 1
    ]
    if (score <= best-score) [
      set best-score score
      set best-strategy ?
    ]
  ]
end

;; this reports a random strategy. a strategy is just a set of weights from -1.0 to 1.0
;; which determines how much emphasis is put on each previous time period when making
;; an attendance prediction for the next time period
to-report random-strategy
  report n-values (memory-size + 1) [1.0 - random-float 2.0]
end

;; This reports an agent's prediction of the current attendance
;; using a particular strategy and portion of the attendance history.
;; More specifically, the strategy is then described by the formula
;; p(t) = x(t - 1) * a(t - 1) + x(t - 2) * a(t -2) +..
;;      ... + x(t - MEMORY-SIZE) * a(t - MEMORY-SIZE) + c * 100,
;; where p(t) is the prediction at time t, x(t) is the attendance of the bar at time t,
;; a(t) is weight for time t, c is a constant, and MEMORY-SIZE is an external parameter.
to-report predict-attendance [strategy subhistory]
  ;; the first element of the strategy is the constant, c, in the prediction formula.
  ;; one can think of it as the the agent's prediction of the bar's attendance
  ;; in the absence of any other data
  ;; then we multiply each week in the history by its respective weight
  report first strategy + sum (map [?1 * ?2] butfirst strategy subhistory)
end

;; In this model it doesn't really matter exactly which patch
;; a turtle is on, only whether the turtle is in the home area
;; or the bar area.  Nonetheless, to make a nice visualization
;; this procedure is used to ensure that we only have one
;; turtle per patch.
to move-to-empty-one-of [locations]  ;; turtle procedure
  move-to one-of locations
  while [any? other turtles-here] [
    move-to one-of locations
  ]
end
; Copyright 2007 Uri Wilensky.
; See Info tab for full copyright and license.
```

Discussion of code:

Firstly a number of global variables and turtle variables are declared. We will discuss how these are used below.

The setup procedure:

Firstly the home-patches and bar-patches global variables assigned Agentsets representing a range of patches in the environment (for display purposes). These are also set to different colours.

The history global variable, stores a list of values indicating the number of agents who attended the bar in the past. Since at the start there is no history we generate a random one. Using the n-values primitive a list of length 2*memory-size is produced of random values between 0..99 (where memory-size is a global variable set on the interface).

The n-values primitive causes the code within the square brackets to be executed 2*memory-size times - each time returning a value and adding it to the end of the list.

The first item in the list represents the most recent attendance, the next item the previous attendance and so on. Note the history is twice the memory size which is required for the evaluation of strategies (see below). The attendance global variable is set to the first element of the history list.

Next a patch at the centre of the bar patches is asked to assign itself (using the self primitive) to the global variable crowded-patch and set its patch label colour to red. This means that any label subsequently assigned the patch will be displayed in red – a label is just a string that gets displayed on the patch. Note the use of the special variable "self" which is always set to the particular agent that is currently executing.

Finally 100 turtles are created. Each is moved to a randomly selected home patch by calling the procedure move-to-empty-one-of and passing in the home-patches Agentset. Note we see here how an Agentset of a particular set of patches can be passed to a procedure.

Each turtle then creates number-strategies (a global supplied by a slider on the Interface) random strategies and stores them in the list called "strategies" (which is a turtle variable):

```
set strategies n-values number-strategies [random-strategy]
```

As before, the n-values primitive causes the code within the square brackets to be executed number-strategies times each time returning a value and adding it to the end of a list. In this case the reporter procedure random-strategy is executed – which returns a single random strategy.

The random-strategy reporter (lower down in the code) actually returns a list of random numbers between -1 and +1:

```
report n-values (memory-size + 1) [1.0 - random-float 2.0]
```

Hence it returns a list of length memory-size+1 random numbers between -1 and 1. Memory-size is a value set in the Interface. This constitutes a strategy in the model (we will discuss below how this is used to make a prediction).

Note this means that the strategies turtle variable becomes a list of lists. It is a list of length number-of-strategies where each item is itself a list that contains memory-size+1 random values (representing a single strategy).

Aside: You can view this list of lists by going to the Interface screen. Click on the setup button. Then in the command center next to the observer> prompt type:

```
show [strategies] of turtle 0
```

This will display a long list on a single line so you need to scroll the widow to see it all. You should see a set of high precision numbers between -1 and +1. Lists have square brackets around them. Hence you should see lists within the list enclosed in square brackets.

You can display each list within the strategies list on a separate line by typing:

```
ask turtle 0 [foreach strategies [ print ?]]
```

The foreach primitive iterates through each item in strategies and executes the commands between the square brackets for that item. The print command displays what follows it. The "?" is substituted by the current item from strategies. In this case it will be a list.

Back to the code on the code screen. After creating the list of strategies the first item from strategies is assigned to the best-strategy turtle variable:

```
set best-strategy first strategies
```

The "first" primitive takes the first item from the list. In this case this will be the first strategy stored in the strategies list. Since strategies have not been evaluated this just initialises the best-strategy to an arbitrary strategy.

Finally each turtle then calls the procedure update-strategies (discussed below) which assesses how well each strategy predicts the past history of attendance and updates best-strategy to that strategy.

The go procedure:

Each time go is called this means it is “bar night” for a new week and each turtle needs to decide if to go to the bar or not based on its prediction of if the bar will be overcrowded (don’t go to the bar) or not (go to the bar). Each turtle then is attempting to predict what all the other turtles will do.

Firstly the crowded-patch (set in the setup procedure) has its label set to an empty string.

Next each turtle is asked to make a prediction using its best-strategy by calling the predict-attendance reporter procedure:

```
set prediction predict-attendance best-strategy sublist history 0 memory-size
```

This line sets the turtle variable “prediction” to a numeric value returned by the predict-attendance reporter. Predict-attendance takes as input a strategy and a history and returns and attendance prediction. In this case the turtle variable best-strategy and a sublist of the history global variable of size memory-size are passed. Hence prediction is set to the prediction made by the current best strategy and history going back memory-size. The way predict-attendance works is discussed in detail later (below).

The “attend?” turtle variable is then set to true if the prediction made is less than or equal to the overcrowding-threshold (a global variable set in the Interface). Hence this records if, given the prediction made, the turtle has decided to attend the bar tonight.

After each turtle has set its “attend?” variable the turtles are asked to move either to a bar patch or a home patch accordingly. Note the line that adds one to the attendance for each turtle attending is not actually required here because it is calculated in the next line.

Next the attendance variable is set to the count of the number of turtles on the bar-patches:

```
set attendance count turtles-on bar-patches
```

The primitive “turtles-on” returns an Agentset containing all the turtles in the patches specified (in this case the bar-patches Agentset).

Next, if attendance exceeds the overcrowding-threshold then the crowded-patch (set in the setup procedure) has its label set to “CROWDED” which is displayed on the screen indicating the bar is overcrowded:

```
if attendance > overcrowding-threshold [
    ask crowded-patch [ set plabel "CROWDED" ]
]
```

Next the history global variable is updated by removing the last item from the end of list and adding the current attendance to the front of the list. The but-last primitive takes a list as input and returns a list which has the last item removed. The fput primitive takes two inputs: an item followed by a list and returns the list with the item added to the front:

```
set history fput attendance but-last history
```

Finally all turtles call update-strategies which causes them to evaluate all of their stored strategies against the new updated history to find which is best and then to assign this to the best-strategy turtle variable.

The update-strategies procedure:

This procedure is the heart of the inductive learning method that each turtle performs to select a best strategy from its list of strategies based on past history.

The basic idea is for the turtle to look back and apply all its strategies to see what they would have predicted over the last memory-size periods if they had been used. These predictions are then compared to the actual attendances and the best predicting overall strategy is selected as the new best-strategy to be used next week by the agent.

The "foreach strategies" loop takes each strategy stored by the agent in turn. Within the foreach loop the current strategy is evaluated by predicting the past memory-size attendances and summing the absolute differences between the actual attendance and the prediction. Hence if a strategy would have got it exactly right for each of the last memory-size attendances then the sum would be zero. The worst possible score is initially assigned to the local variable best-score.

To derive the prediction for the last memory-size attendances from the history for the current strategy a "repeat memory-size" loop is performed which iterates the code block that follows for memory-size times increasing a counter called week from 1 to memory-size.

The first line within the code block performs a prediction for week-1 using the current strategy and the previous memory-size history before that week. Note that weeks go backward in time. Week 0 means the most current attendance. Week 1 the previous attendance and so on.

```
set prediction predict-attendance ? sublist history week (week + memory-size)
```

The prediction variable stores the prediction produced by the predict-attendance reporter (defined below) which takes as input a strategy and a history and reports the prediction of next attendance made by that strategy given that history.

Here the "?" represents the current strategy in the foreach loop. The sublist primitive takes three inputs: a list, a start position and an end position. It reports the sublist from the start to the end position (without including the item at the end position). Hence when week=1, and, say, memory-size=5 then we would get a sublist of history ranging from week 1 to week 5 and the prediction produced would be for week 0. If week=2 then we get a sublist from week 2 to week 6 and the prediction is for week 1.

After the prediction is made the difference between the prediction and the actual attendance is added to the score (the higher the score the worse the prediction):

```
set score score + abs (item (week - 1) history - prediction)
```

The term "item (week – 1) history" returns the item stored at position week-1. That is the attendance that the prediction attempted to predict. Prediction is subtracted from this and the function "abs" applied (which gives an absolute value, that is a positive valued difference meaning any negative value becomes a positive value).

Hence score is increased by how far the prediction was from the actual attendance.

Following this the week local variable is increased by 1 and the repeat loop cycles again until memory-size iterations. By increasing week each time successive iterations of the repeat loop go back in history trying to predict the historical attendances based on the previous history of that attendance and accumulating the error in the score variable.

After the repeat loop the score is tested to see if it is currently the best (lowest) obtained score so far. If so then it is stored (in the local variable best-score) along with the strategy (in the best-strategy turtle variable).

Next the foreach loop cycles again, considering the next strategy, until all strategies stored in the turtle have been considered.

Hence by the end of the update-strategies procedure the best-strategy turtle variable will contain the current best-strategy as assessed by evaluating every strategy against memory-size of past history.

<u>The predict-attendance procedure:</u>

This is the core of the entire model since it is used by turtles to both decide how to act (attend the bar or not – called by the go procedure) and to evaluate strategies to find the best (called from the "update-strategies" procedure).

It returns a prediction of the next attendance given a strategy and a history. Strategy is a list of length memory-size+1 of weights. Each weight is a number between -1 and +1. History is a list of past attendances at the bar.

The prediction is formed by treating the first weight as a constant and multiplying each subsequent weight by the associated past attendance and summing them along with the constant (it's a kind of simple regression model). This is accomplished in a single line by using the "map" privative:

```
report first strategy + sum (map [?1 * ?2] butfirst strategy subhistory)
```

Map can be applied in several contexts. Here map takes two lists (of the same length) as input and produces a new list composed of the product of the associated items (those in the same position) in the input lists.

Aside: To get an idea of how map works go to the Interface screen. In the command center next to the observer> prompt type:

```
print (map [?1 * ?2] [1 2 3] [2 1 5])
```

You should see as output a list which is the product of the items in the two lists:

```
[2 2 15]
```

Note: map can also be used on a single list or more than two lists. For example, type:

```
print (map [round ?] [0.4 0.5 1.6])
```

Round is a primitive that rounds a number to the nearest whole number. You should see:

```
[0 1 2]
```

Here is an example for three lists which adds the corresponding item in each list together, type:

```
print (map [?1 + ?2 + ?3] [1 2 3] [4 3 2] [5 6 7])
```

giving:

```
[10 11 12]
```

Here is a more complex example combining foreach with map. Click the setup command on the Interface to initialise a new turtle population. In the command center type:

```
ask turtle 0 [foreach strategies [print map [precision ? 2] ?]]
```

This should display all the strategies of turtle 0 (one on each line) with each item in the strategy lists displayed to 2 decimal places of precision. Note the first "?" refers to items from the list which follows. That list is represented by the second "?" which refers to items from the strategies list of the foreach loop (this makes the line confusing to read!).

Back in the code the first list used as input to the map command is the strategy without the first item (the butfirst primitive returns a list without the first item) because this is a constant. The second list is the subhistory of attendances passed as input to the predict-attendance procedure.

Hence map in this context returns a list of products of the weights and the attendances. These are them summed with the sum function and added to the first item from strategy (which is the constant).

This becomes the prediction for the next attendance which is reported back from the procedure using the report primitive.

Task: Modify the model so the following is displayed on the interface when it is executed: a) average attendance per cycle; b) proportion of cycles in which the bar is crowded (where 1 is always and 0 is never); c) proportion of correct actions made by turtles over all cycles. A correct action is staying at home if the pub was crowded or going to the pub if it was not crowded.

Task: To further analyse correct actions add a histogram that displays the distribution of correct actions over all turtles for all cycles.

Task: Examine the comments in the code before the predict-attendance procedure. Notice that in the comments the constant should be multiplied by 100 but this is not done in the implementation. Correct the implementation and examine if it effects the behaviour of the model. Hint: You need to change update-strategies too – can you determine where and what you need to change?

Task: The El Farol Bar model relies on agents selecting from fixed strategies. Modify the program so that turtles replace their worst performing strategy with a new random one at end of the update-strategies procedure.