Introduzione alla modellizzazione ad agenti con NetLogo 4

- Lists, Links, 3D…

Franco Bagnoli 2017

Liste

NetLogo (come tutti i linguaggi derivati da Logo) si ispira molto al lisp, ovvero alla manipolazione di liste (anche se purtroppo in NetLogo ci sono liste, stringhe e e agentset che condividono molte funzioni ma non tutte). Provare con test-list.nlogo

Ricordarsi gli spazi attorno agli operatori!!!!

operator
remove-item number list

argument
6

thelist
2 5 8 9 [1 2] [3 4] "a"

Dammi un esempio!

string to be executed
remove-item 6 [2 5 8 9 [1 2] [3 4] "a"]

results
[2 5 8 9 [1 2] [3 4]]

Liste

il codice è un po’ “truccoso” per far sì che gli esempi cambino continuamente

prima parte:

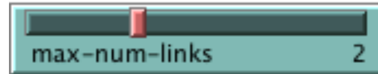
```
globals [str op]

to-report myreporter [string]
  let i position " " string
  set op (substring string 0 i)
  set str (word op " " argument " [" thelist "]")
  report  str
end

to-report to-string [alist]
  report ifelse-value (empty? alist) [
    ""
  ][
    ifelse-value ((length alist) = 1) [
      (word item 0 alist)
    ][
      reduce [[x y] -> (word x " " y)] alist
    ]
  ]
end
```

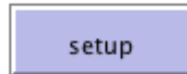
Liste

second
a parte

```
to example
  ; some random list
  let complete-list [1 2 3 4 5 6 7 8 9 [1 2] [3 4] [4 5] "\"a\"" "\"b\"" "\"c\""]
  let sub-list n-of 7 complete-list
  let alist to-string(sub-list)
  let help  (list
    (list "but-first" "" alist)
    (list "but-last" "" alist)
    (list "empty?" "" to-string(n-of (random 2) n-of 5 complete-list))
    (list "filter" "[x -> ifelse-value is-number? x [true][false]]" alist)
    (list "first" "" alist)
    (list "fput" "\"b\"" alist)
    (list "item" "3" alist)
    (list "last" "" alist)
    (list "length" "" alist)
    (list "lput" "7" alist)
    (list "map" "[x -> x * x]" to-string(map [x -> random 100] range 5))
    (list "max" "" alist)
    (list "member?" (word one-of complete-list) alist)
    (list "min" "" alist)
    (list "modes" "" to-string(sentence n-of 3 sub-list n-of 3 sub-list n-of 3 sub-list))
    (list "n-of" "3" alist)
    (list "position" (word one-of sub-list)  alist)
    (list "one-of" "" alist)
    (list "range" "" "0 20 3")
    (list "reduce" "[[x y] -> x * y]" "1 2 3 4 5")
    (list "remove" (random length sub-list) alist )
    (list "remove-duplicates" "" to-string(sentence n-of 3 sub-list n-of 3 sub-list n-of 3 sub-list))
    (list "remove-item" (word random length sub-list) alist )
    (list "reverse" "" alist)
    (list "shuffle" "" alist)
    (list "sort" "" alist)
    (list "sort-by" "[[x y] -> (item 0 x) < (item 0 y)]" "[\"c\" 3] [\"a\" 1] [\"b\" 2] ")
  )
  let thehelp item 0 (filter [x -> (item 0 x) = op] help)
  set argument  item 1 thehelp
  set thelist item 2 thehelp
end
```

Links





I link connettono due turtle. La cosa interessante dei link è che possono comportarsi come molle e quindi aiutare a disporre i nodi.

Proviamo a scrivere un programma che crea un grafo (graph.nlogo)

Links

```
breed [nodes node]

to setup
  clear-all
  set-default-shape nodes "circle"
  ;; create some random network
  create-nodes 30 [ set color blue ]
  ask nodes [
    create-links-with n-of (random max-num-links + 2) other nodes
  ]
  ;; lay it out so links are not overlapping
  repeat 500 [ layout ]
  ;; leave space around the edges
  ask nodes [ setxy 0.95 * xcor 0.95 * ycor ]
  ;; put some "walker" turtles on the network
  reset-ticks
end

to layout
  layout-spring nodes links 0.5 2 40
end
```

Il codice è molto semplice. Si può notare la funzione n-of che prende un certo numero di oggetti da una lista, e il fatto che il layout è ripetuto 500 volte per essere sicuri di aver raggiunto la distribuzione asintotica.

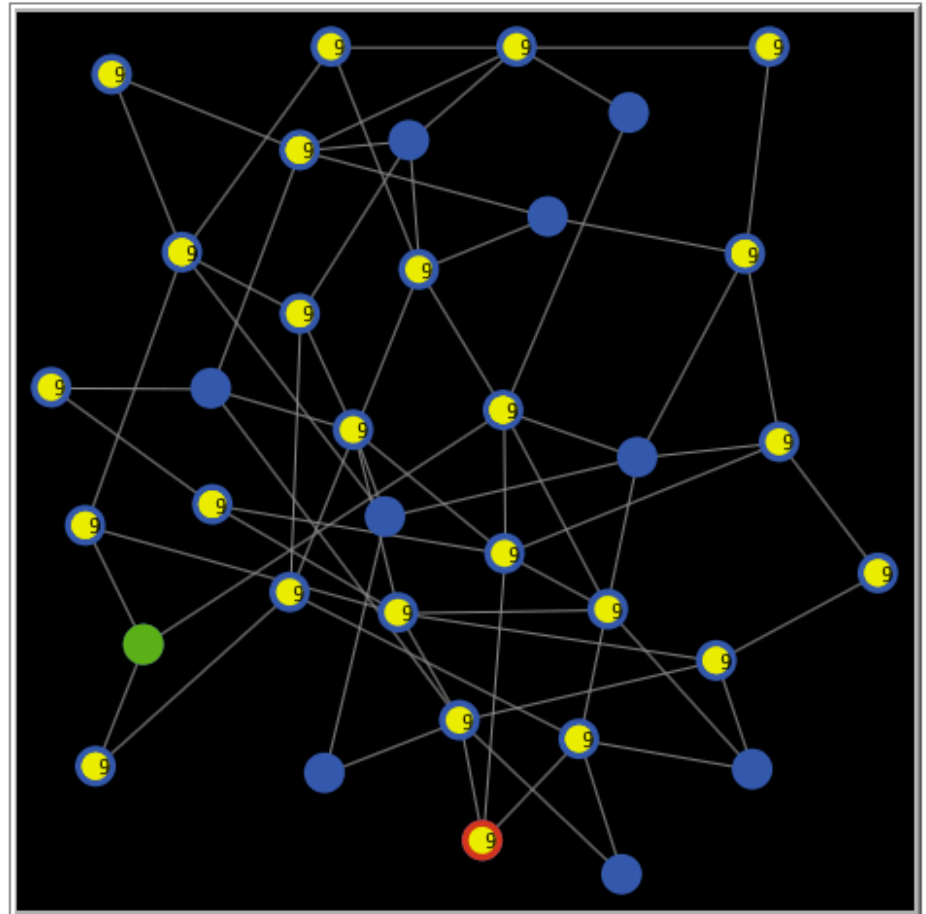
Links

setup

num-died-walkers
3

average length
6.333333333333333

min-time
5



Possiamo adesso passare a qualcosa di più serio: una sorgente che emette camminatori (walkers) che avanzano a caso (incrementando un contatore) finché non cadono in un pozzo (graph-walhers.nlogo)

Links

Nel codice (prima parte_ si usano due tipi di breed: node e walker

```
breed [nodes node]
breed [walkers walker]
globals [num-died-walkers sum-time-died-walkers min-time well]

walkers-own [counter location] ; location is one of nodes

to setup
  clear-all
  set-default-shape nodes "circle"
  set-default-shape walkers "circle"
  ;; create some random network
  create-nodes num-nodes [
    set color blue
    set size 1.5
  ]
  ask nodes [
    create-links-with n-of (random (max-num-links - 2) + 2) other nodes
  ]
  ;; lay it out so links are not overlapping
  repeat 500 [ layout ]

  ; source of walkers
  ask one-of nodes [
    set color green
    set size 1.5
    let tmp self ; used to pass starting location to walkers
    hatch-walkers num-walkers [
      set color yellow
      set size 1
      set location tmp
    ]
  ]
  ; well
  ask one-of nodes [
    set color red
    set size 1.5
    set well self
  ]
  set min-time 100000
  reset-ticks
end
```

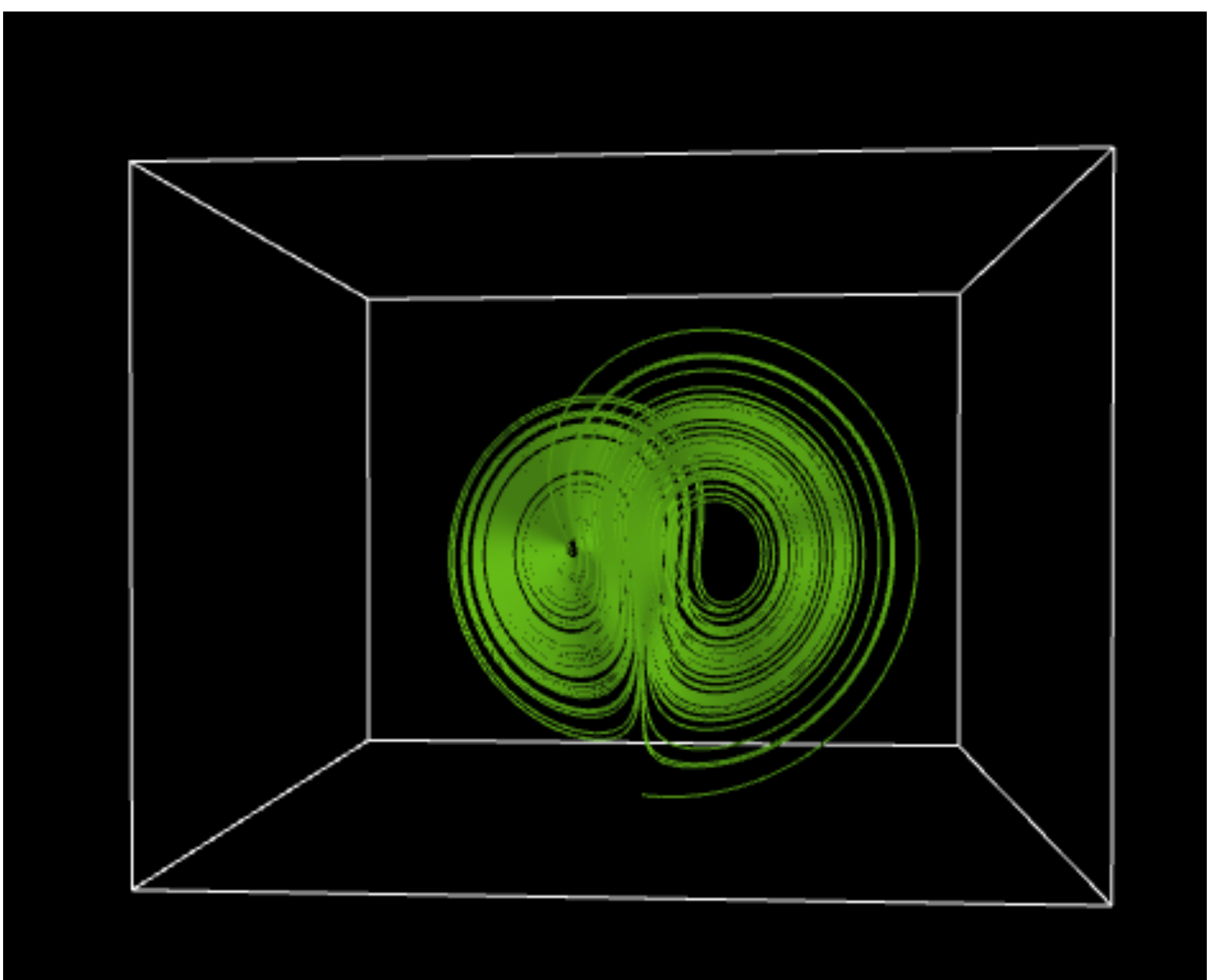
Links

Seconda parte. Si noti come si usi un nodo come variabile di un altro breed

```
to layout
  layout-spring nodes links 0.5 2 40
end

to go
 no-display
  ask well [
    ask walkers-here [
      set num-died-walkers num-died-walkers + 1
      set sum-time-died-walkers
          sum-time-died-walkers + counter
      if counter < min-time [
        set min-time counter
      ]
      die
    ]
  ]
  ask walkers [
    let new-location one-of [link-neighbors]
       of location
    face new-location
    move-to new-location
    set location new-location
    set counter counter + 1
    set label counter
    set label-color black
  ]
  display
  tick
end
```

Lorenz 3D

NetLogo esiste anche in versione 3D, che però è difficile da usare perché non si capisce nulla.



Un buon uso del 3D è quello di visualizzare traiettore caotiche come nel Modello di Lorenz.

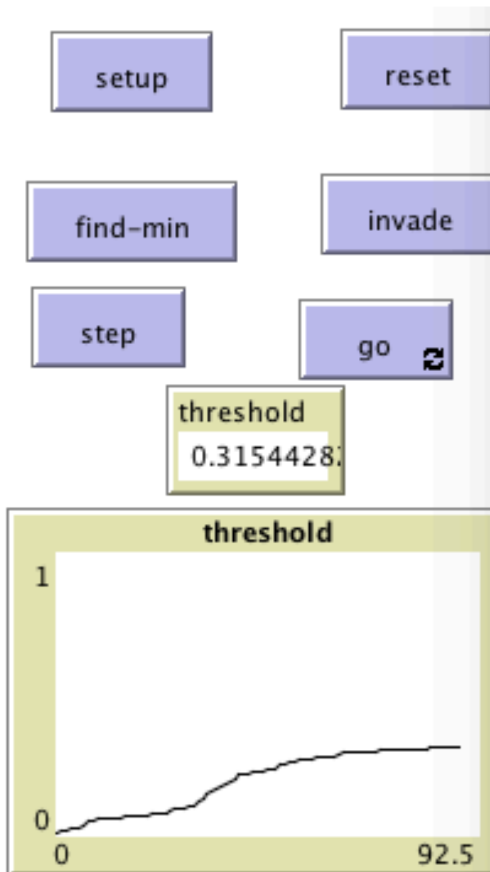
Lorenz 3D

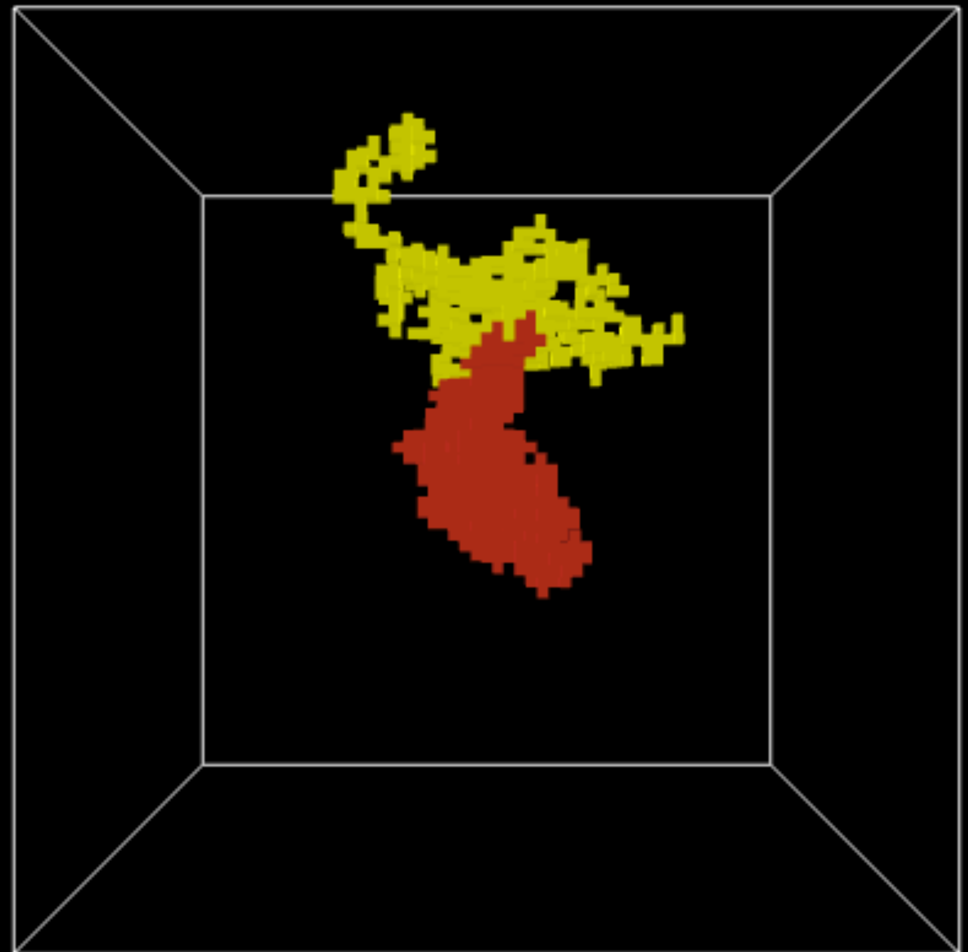
Il codice è molto semplice

```
to setup
  clear-all
  set dt 0.001
  set x .04
  set y -.1
  set z .3
  set s 10
  set r 28
  set b 8 / 3
  create-turtles 1 [
    setxyz x y z
    set size 0
    set color green
    pen-down
  ]
end

to go
  let x1 x
  let y1 y
  let z1 z
  set x x1 + s * (y1 - x1) * dt
  set y y1 + ((r * x1) - (x1 * z1) - y1) * dt
  set z z1 + (x1 * y1 - (b * z1)) * dt
  setxyz x y z
end
```

Invasion percolation 3D

Invasion percolation 3D (un modello per la crescita tumorale?)





Invasion percolation 3D

prima parte

```
globals [threshold continue? stop?]

patches-own [resistance]

to setup
  clear-all
  ask patches [
    set resistance random-float 1
  ]
  ask patch 0 0 0 [
    set pcolor yellow
    set resistance 0
    ; avoid sudden invasion by lowering the neighboring resistance
    ask neighbors [
      set resistance random-float 0.1
    ]
  ]
  reset
end

to reset
  reset-ticks
  clear-plot
  reset-timer
  set stop? false
  set threshold 0
  ask patches [
    set pcolor black
  ]
  ask patch 0 0 0 [
    set pcolor yellow
  ]
end
```

Invasion percolation 3D

seconda parte

```
to find-min
  ; marks in red the cells on the periphery
  foreach  [neighbors6] of patches with [pcolor = yellow] [
    x -> ask x [
      if  pcolor = black [
        set pcolor red
      ]
    ]
  ]
  ; looks for the cell with minumum resistance in the periphery
  set threshold min [resistance] of patches with [pcolor = red]
  ask patches with [pcolor = red and  resistance <= threshold] [
    set pcolor green
  ]
  plot threshold
end

to invade
  set continue? true
  while [continue? = true] [
    set continue? false
    myloop
  ]
end
```

Invasion percolation 3D

terzaparte

```
to myloop
  ; stop condition
  ifelse not any? patch-set ([neighbors6] of patches with
    [pcolor = yellow and
    (pxcor = min-pxcor or
          pxcor = max-pxcor or
          pycor = min-pycor or
          pycor = max-pxcor or
          pzcor = min-pzcor or
          pzcor = max-pzcor )
  ]) [
    foreach  ([neighbors6] of patches with [pcolor = yellow]) [
      x -> ask x [
        if  pcolor != yellow and resistance <= threshold [
          set pcolor yellow
          set continue? true
        ]
      ]
    ]
  ][
    set stop? true
  ]
end

to go
  find-min
  invade
  if stop? [stop]
end
```