Boris

NetLogo thin agent-to-agent communications link tutorial

---

outline

This tutorial describes the nlbloris extension for NetLogo which enables Netlogo agents to send messages to/from each other.

The nlboris extension provides a *thin* NetLogo communications link – ie: it only allows NetLogo agents to communicate with each other, not with external agents. A *thicker* communications link which allows NetLogo agents to exchange messages with agents written in Java, Lisp, C# is under development, for updates see www.agent-domain.org.

examples

The following guide is based around 3 examples of NetLogo models. These models are designed to demonstrate the use of the nlboris extension – it is quite possible to build these models without using agent-to-agent messaging (particularly the first 2).

example 1 – single target & follower

There are 2 turtles in this example (i) the leader or *target* (colored green in the model) and (ii) the follower (blue). The aim of the follower is to reach the target.

The model could easily be built by allowing the follower to detect the target and move towards it but for the sake of example we use a different approach which works as follows...

1. the target moves around, occasionally changing direction (its direction of movement does not consider the position of the follower);
2. at each move the target sends a message to the follower to tell the follower where the target has moved to;
3. at each step, the follower receives a message (from the target) and moves towards the location specified in the message.

key aspects of model code

These notes assume that you have access to the full model code – please refer to the model as you read what follows.

in declarations section

- `extensions [nlboris]`
  this loads the NetLogo extension which handles agent messaging

- `globals [portal]`
  declares a variable to hold the communications hub (Boris calls communications hubs "portals" but you can name the variable anything you like)

in setup-globals

- `set portal nlboris:make "POP"`
  make a new communications hub/portal called "POP" and assign it to the *portal* variable. All portals have names, these names are not used in the *thin* link described here but are required if they are connected to external agents

in setup-followers

- `nlboris:register portal "follower"`
  register the follower with the comms link, the arguments provided are (i) the comms link (ii) a name for the agent ("follower") which will be the name used for sending & receiving messages. Note: if turtles are not registered they will not be able to exchange messages.

in setup-targets

- `nlboris:register portal "target"`
  as above... register the target turtle as a communicable agent called "target"

in move-targets

- `nlboris:send-msg portal "follower" (list xcor ycor)`
  send the agent registered with the name "follower" a message with the target's *xcor* and *ycor*. Messages are sent as NetLogo Lists.

in move-followers

most of the interesting work is done in the move-followers procedure. This procedure also contains the type of code fragments that we will often use for agents which receive messages, it is examined further in the next few sub-sections. move-followers is specified as follows...

```
to move-followers
  ask followers
  [ if nlboris:is-msg-waiting portal
    [ let m nlboris:get-msg portal

      let from item 0 m
      let msg  item 1 m
      let tx   item 0 msg
      let ty   item 1 msg

      facexy tx ty
      fd 1
      ask targets-here [die]
    ]
  ]
end
```

stage-1 – receiving messages

The code fragment...

```
if nlboris:is-msg-waiting portal
[ let m nlboris:get-msg portal
  ...process message...
]
```

...is quite typical, it checks to see if there is a message waiting & if so gets that message...

- `nlboris:is-msg-waiting portal`
  see if there is a message waiting for this turtle

- `let m nlboris:get-msg portal`
  get the message & store it in a (local) variable ***m***

stage-2 – message format

All messages are provided in the form of a List. The first element of the list is the name of the agent who sent the message, the second element is the main content of the message (ie: what was sent). This second element is always in the form of a sub-list.

The code fragment...

```
let from item 0 m
let msg  item 1 m
let tx   item 0 msg
let ty   item 1 msg
```

...pulls the received message apart. The first two lines get the name of the sending agent (item 0 of m) and then the main message content (item 1 of m). The next two lines (greyed out) extract information from the message content – they are specific to this example.

example 2 – multiple followers

Similar to the last example – there are 2 breeds (targets & followers). As before there is one *target* (colored green in the model) but in this example we use multiple followers (blue). As in the last example the followers aim to reach the target.

The approach is as follows...
1. the target moves around, occasionally changing direction;
2. at each move the target broadcasts a message to all followers to tell them its position;
3. each follower receives the message sent by the target and moves towards the location specified in the message.

key aspects of model code

The code used to specify this model is almost the same as in the last example except that the target *broadcasts* its message to all followers instead of just sending it to the one & only follower. This is specified as...

- `nlboris:broadcast portal "FOLLOWERS" (list xcor ycor)`

Note a couple of things about this...

- the use of *broadcast* is similar to send-msg except that a breed name is used as the recipient of the message instead of an agent name;
- breed names are in upper-case;
- breed names are in plural.

Followers each have their own name (duplicate names should not be used in nlboris), we make unique names by adding the *who* number of Netlogo turtles/breeds to a standard name. Like this...

- `nlboris:register portal (word "follower" who)`
  so a turtle with a who number of 32 will be registered as "follower32"

example 3 – mine clearance

The scenario is this... the environment contains multiple mines and one bin. Mines can be made harmless by putting them in the bin (don't try this at home). When all the mines are in the bin the model stops running.

There are two types of agent: *detectors* and *cleaners*. Detectors have sensors which can give the direction of a mine but they can only determine a mine's exact position by moving up to it. Detectors are not capable of picking up mines so cannot carry them to the bin.

Cleaners have no sensing capabilities so cannot locate mines on their own but they are able to pick up mines (one at a time) & carry them to the bin.

Detectors and cleaners work together: detectors find mines then send messages to cleaners to tell the cleaners where the mines are located. Cleaners get messages from detectors containing information about the location of mines then go pick up those mines and carry them to the bin.

Note: this is roughly similar to an example used in a training session I once attended for building BDI agents in 3APL so the credit for the nature of this example goes to the authors of the 3APL training materials. Our model is quite different to the kind of solution you would build in 3APL (we are not using BDI agents) but the example is nice to use for investigating agent messaging.

running the model

The model allows users to specify the number of mines, detectors & cleaners. The bin is in the middle of the environment (xcor=0, ycor=0) and all of the agents start there. Mines are randomly scattered around the environment.

Mines are circular black & white blobs, the bin is a green garbage-can shape, detectors are blue (standard turtle shape), cleaners are yellow when not carrying a mine and red when they are carrying.

When the model starts running, detectors (blue) leave the bin area, each heading for a mine. When a detector reaches a mine it selects a cleaner (at random) and sends it a message about the mine's location. The cleaner (yellow) then leaves the bin area to fetch the mine and the detector moves off towards another mine.

When the cleaner reaches a mine the cleaner turns red and the mine symbol disappears from the visual environment, the cleaner then takes the mine to the bin. When a (red) cleaner holding a mine reaches the bin the cleaner turns yellow & checks if it has a message waiting from a detector about another mine that needs putting in the bin.

Notice (i) the detectors inevitably finish their work before the cleaners (ii) some cleaners do more work than others because the detectors randomly choose which cleaners to inform about mines (so some finish before others).

key aspects of model code

- the bin is implemented as the only member of a turtle breed but it is not active;
- mines are also implemented as members of a breed – this makes a couple of things easier but they are not active either;
- breed.status variables are used to keep track of the current state and/or objective of turtles/breeds;

status options for different breeds...

- detectors can be either...
  (i) tracking – they have detected a mine and are moving towards it to get its coordinates (they spend most of their time doing this), or
  (ii) looking – they are detecting the direction of a new mine to track;

- cleaners can be...
  (i) waiting – idle with nothing to do until a message arrives from a detector
  (ii) fetching – moving towards a mine that they have been told about by a detector
  (iii) carrying – carrying a mine back to the bin;

- mines can be...
  (i) hidden – they have not been detected yet
  (ii) found – they have been detected and a detector is now tracking them
  (iii) to-clean – a detector has located their exact position & informed a cleaner but the cleaner has not picked them up yet
  (iv) cleaned – a cleaner has them but they are not in the bin yet.

nlboris use

While the strategy/purpose of communication is different, the use of nlboris is similar to the previous models, ie:

in the setup phase

set up a portal

- `set portal nlboris:make "P1"`

register each detector

- `nlboris:register portal (word "detect" who)`

register cleaners, note that cleaners know their names

- `set name (word "clean" who)`
- `nlboris:register portal name`

in move-detectors

Once a detector has reached a mine it knows the mine's position. It sends this information to a randomly selected cleaner.

- `let cleaner [name] of (one-of cleaners)`
- `nlboris:send-msg portal cleaner (list ...`*`mine coordinates...`*`)`

in move-cleaners

The code here follows that used by followers in the previous example.