Appendix C

The ql-extension

Since the ql-extension is an extension of NetLogo, the latter must be installed first. The installation was tested for NetLogo 5.2.1. Afterwards, a directory named `ql` should be created in the `extensions` subdirectory of the `NetLogo` installation (see also `http://ccl.northwestern.edu/netlogo/docs/extensions.html`). All files from

```
https://github.com/JZschache/NetLogo-ql/tree/master/extensions/ql
```

should be downloaded and moved to the newly created directory `extensions/ql`. For example:

```
git clone https://github.com/JZschache/NetLogo-ql.git
mv NetLogo-ql/extensions/ql path-to-netlogo/extensions
```

After starting NetLogo, a sample model from `NetLogo-ql/models` can be loaded.

C.1 A first example

Amongst other things, the ql-extension enables the simulation of agents who make decisions by melioration learning. As explicated in chapters 6 and 7, this algorithm can be applied in situations of repeated decision-making. The set of choice alternatives should be relatively stable. A simple example is given in the NetLogo model of listing C.1 and figure C.1 (downloadable as `NetLogo-ql/models/basic.nlogo`).

```
extensions[ql]

turtles-own[exploration-rate exploration-method
            alternatives q-values frequencies rel-freqs]

to setup
  clear-all
  create-turtles n-turtles [
    setxy random-xcor random-ycor
    set exploration-rate global-exploration
    set exploration-method "epsilon-greedy"
    set alternatives [ 0 1 ]
    set rel-freqs [0 0]
  ]
  ql:init turtles
  reset-ticks
end

to go
  ask turtles [
    let action ql:one-of [ 0 1 ]
    ifelse (action = 0) [
      fd 1
      ql:set-reward action (random-normal forward-reward 1)
    ] [
      right 90
      ql:set-reward action (random-normal right-reward 1)
    ]

    let total-freq sum frequencies
    if (total-freq > 0) [
      set rel-freqs map [? / total-freq] frequencies
    ]
  ]
  tick
end
```

Listing C.1: NetLogo code of first example

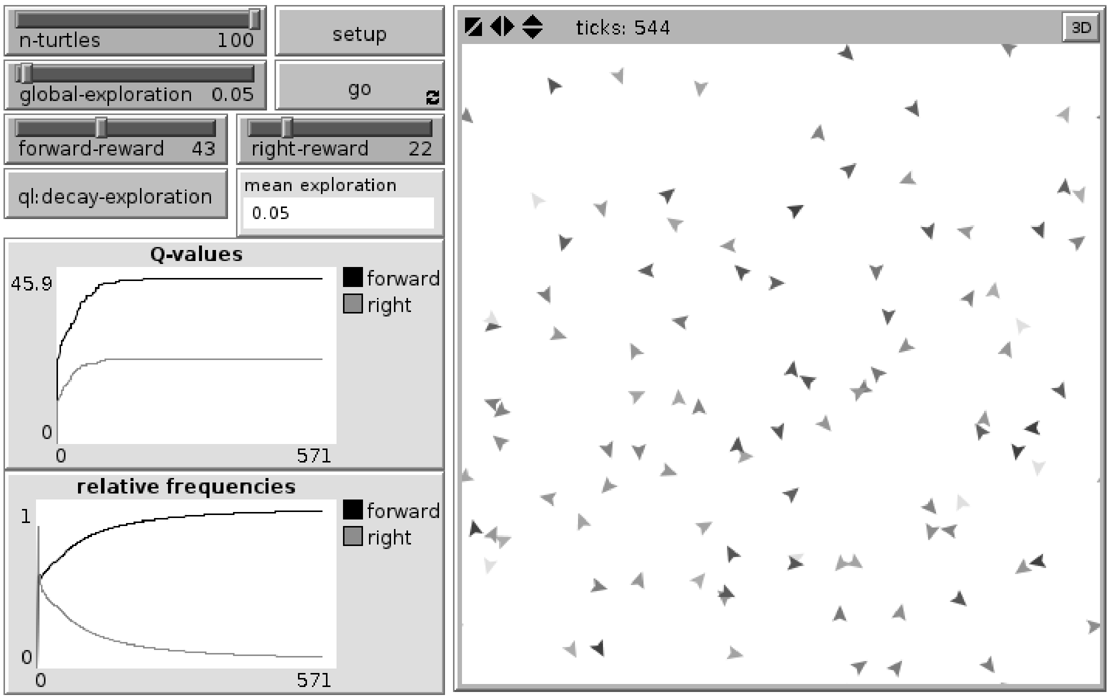


Figure C.1: NetLogo interface of first example

The agents of this model (the turtles) learn Q-values of two alternatives: "move one step forward" (alternative 0) and "turn to the right" (alternative 1). The reward of either alternative is drawn from a normal distribution with mean `forward-reward` or `right-reward` and standard deviation one. Besides these means, also the number of turtles (`n-turtles`) and the initial exploration rate (`global-exploration`) are set by the NetLogo interface (see figure C.1).

After including the line `extensions[ql]` at the beginning of the NetLogo code, the ql-extension is ready for use. Figure C.1 and listing C.1 contain a number of special commands and reporters, which are identified by the prefix '`ql:`'. In the `setup` procedure of listing C.1, turtles are created and randomly distributed over the NetLogo world. The ql-extension is initialised by `ql:init` and an agentset (a turtleset or a patchset). If any of the variables `exploration-rate`, `exploration-method`, or `alternatives` are specified before `ql:init` is called, these values are used for the agents. Otherwise, default values are employed (0.05, "epsilon-greedy", and empty list). The default values as well as the names of the variables are defined in the configuration file `application.conf`.

The *list of alternatives* must be a list of integers. The *exploration rate* is a positive number. Note that this rate cannot be interactively changed during the simulation (as it is usually possible in NetLogo). With `ql:decay-exploration`, a decay of this rate can be started at any point during the simulation. Afterwards, the initial exploration rate is divided by the logarithm of the number of choices (it starts counting the choices with the call of `ql:decay-exploration`).

Three *methods of exploration* are currently implemented:

- "epsilon-greedy" denotes the implementation of algorithm 5.2.1, but with exploration decreasing at logarithmic speed if `ql:decrease-exploration` is called: $\varepsilon_s \leftarrow \frac{\varepsilon}{\log\left(2+\sum_{j\in E} K_t(s,j)\right)}$.
- "softmax" refers to Boltzmann exploration as described in section B.1. The temperature is given by the exploration rate and decreases logarithmically if `ql:decrease-exploration` is called.
- "Roth-Erev" stands for the model of algorithm 6.2.2. The exploration rate decreases logarithmically if `ql:decrease-exploration` is called.

As stated in sections 5.2.2 and 6.2, the learning algorithms require the agents to use and modify Q-values when making decisions. The current state of the Q-values are accessed via the agent variable `q-values`, which is a list of numbers. This list is automatically updated during the simulation if defined by `turtles-own` (or `patches-own`). Besides the Q-values, also the names of the alternatives (`alternatives`), the frequencies of choice (`frequencies`), and the exploration rate (`exploration-rate`) are continuously updated. The names of the variables can be changed in the configuration file (`application.conf`). The relative frequencies `rel-freqs` are not part of the ql-extension and must be implemented separately (see listing C.1).

The decision of an agent and the assignment of a reward are controlled by `ql:one-of` and `ql:set-reward`:

- `ql:one-of` takes a list of alternatives (integers) and returns one of them by employing the specified exploration method.
- `ql:set-reward` maps a reward to a decision.

Furthermore, the *state* of the environment is considered by the agents (see section 5.2). The agent variable `state` keeps track of the state if it is defined by `turtles-own` (or `patches-own`). The state is an integer and can be set for each agent similar to the other variables (e.g. `alternatives`) before calling `ql:init`. Otherwise, the default (0) is used. A change in state is executed by calling `ql:set-reward-and-state` instead of `ql:set-reward` and appending a third parameter (the new state). Since the agent distinguishes between the states, the list of `alternatives` becomes a list of pairs of integers after `ql:init` was called. The first element of the pair indicates the state and the second element the alternative.

C.2 Parallelising the simulation

By building on the Akka framework, the ql-extension is able to parallelise the simulation and utilise multiple cores (Wyatt, 2013). Akka is written in Java and Scala. Since concurrency in Java is based on threads, also Akka uses threads. But the difficulties of data sharing and synchronisation are handled by a message-passing architecture. More concretely, Akka requires the implementation of "Akka actors" that run independently and share data by sending messages to each other (for a general introduction to the differences between thread-based and message-passing parallel programming, see Rauber and Rünger, 2013).

In the basic example of the previous section, the simulation is already parallelised. First, the NetLogo threads are not used for the ql-extension, which means that the latter runs independently of the former. Second, the learning and decision-making of the agents take place simultaneously because the ql-extension runs on multiple threads. The number of threads is controlled by the configuration file (`application.conf`: `akka.actor.default-dispatcher`; see also `http://doc.akka.io/docs/akka/2.0.5/general/configuration.html`).

Nevertheless, many parts of the simulation are executed by NetLogo, which does not parallelise naturally. This is a major bottleneck of the simulations because the ql-extension must wait for NetLogo to finish its calculations. The ql-extension solves this problem by the operation of multiple concurrently running instances of NetLogo. The deployment of multiple NetLogo instances is enabled by setting `enable-parallel-mode` to `true` (`application.conf`).

Given the current implementation, certain conditions must hold for the parallel mode to work:

1. It must be possible to assemble the agents into several groups. This may happen once at the beginning of the simulation (static groups) or repeatedly at each round (dynamic groups).

2. The situation must permit the rewards to be calculated for each group separately at a given point in time. Only the decisions of the group members and global variables of NetLogo can be used for this calculation.

This impedes the usage of the parallel mode, for instance, in the foraging model of chapter 8. The reward of an agent cannot be calculated without considering all other agents and the current state of the environment. Hence, all agents must be member of the same group, and no parallelisation is possible.

For a better understanding of the parallel mode, the architecture of the software is explained in the next section. It can be skipped if only the usage of the simulation is relevant. Section C.4 contains an example that makes use of the parallel mode.

C.3 The architecture of the ql-extension

The architecture of the ql-extension is illustrated by the class diagram of figure C.2. It clarifies the connection between the extension and the NetLogo package `org.nlogo`. It also explains how concurrency is implemented by "Akka actors". First, each NetLogo agent (a turtle or a patch) is linked to an "Akka actor". This is realised by the `QLAgent` class, which constitutes the counterpart of a NetLogo agent in the ql-extension. It is characterised by an exploration rate, a list of `QValues`, and a decision-making algorithm ("epsilon-greedy", "softmax", or "Roth-Erev"). A `QValue` instance is created for each alternative and specifies the current Q-value. The decision-making algorithm returns an element of a list of alternatives (a list of integers). It uses the exploration rate and the `QValues`.

Agents are grouped together by the class `NLGroup`. This is a subclass of `org.nlogo.api.ExtensionObject`, which makes it accessible within NetLogo code. It consists of NetLogo agents and the corresponding `QLAgents`.

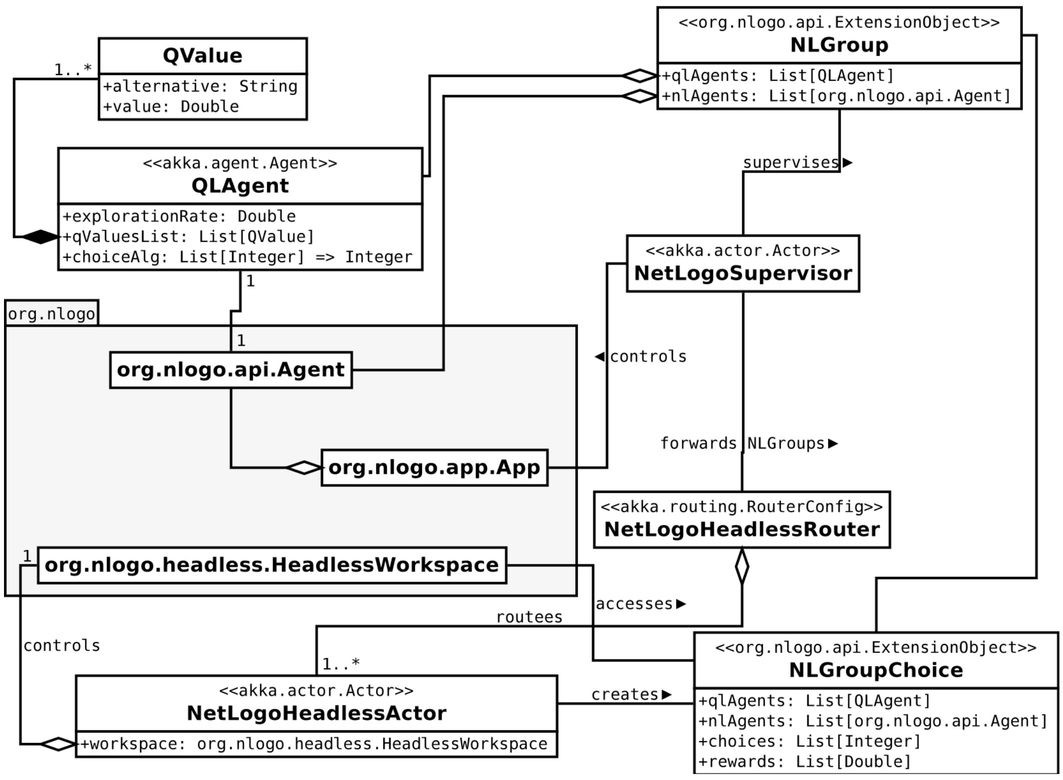


Figure C.2: Class diagram of the ql-extension

The main "Akka actor" of the extension is the `NetLogoSupervisor`. There is only one instance of this class. The `NetLogoSupervisor` has mutliple tasks. For example, it supervises all `NLGroups` and continuously triggers the choices of the agents. The speed of the repeated trigger is regulated by the corresponding slider of the NetLogo interface. When triggering the choice of the agents, the list of all `NLGroups` is forwarded to the `NetLogoHeadlessRouter`. Depending on the number of `NetLogoHeadlessActors`, the router splits this list in multiple parts. Afterwards, the `NetLogoHeadlessActors` handle the choices of the agents, and the `NetLogoSupervisor` is free to do other things.

More specifically, the `NetLogoSupervisor` also controls the main NetLogo instance (`org.nlogo.app.App`). On the one hand, it repeatedly calls the command `update` after all agents have received a reward. This command can be used to set a new `tick` and, hence, to update the NetLogo interface. On the other hand, the `NetLogoSupervisor` invokes the main NetLogo instance if the groups change during the simulation (section C.7).

When initialising the `NetLogoSupervisor` by `ql:init`, several *headless* workspaces of NetLogo are started in the background. *Headless* means that no graphical user interface is deployed. The number of headless workspaces is specified in the configuration file (`application.conf`). A separate "Akka actor" (the `NetLogoHeadlessActor`) controls each headless NetLogo instance. This actor continuously receives a list of `NLGroups` from the `NetLogoSupervisor` (via the `NetLogoHeadlessRouter`) .

The headless NetLogo workspaces and the `NetLogoHeadlessActors` were added to the ql-extension in order to improve the performance. The interaction with NetLogo is the main bottleneck of the ql-extension. But repeatedly invoking NetLogo is necessary because the reward function, which maps the agents' choices to rewards, should be specified in the NetLogo model and not within the ql-extension. Therefore, multiple instances of NetLogo are run in parallel. Their only task is to repeatedly calculate the rewards of several groups of agents.

The performance of repeatedly calling the reward function is optimised by compiling this function only once. This is problematic because the NetLogo extensions API does currently not support the passing of arguments to a compiled function (see `https://github.com/NetLogo/NetLogo/issues/413`). A solution was mentioned by Seth Tisue in the corresponding discussion[1] and is implemented in the ql-extension. Each `NetLogoHeadlessActor` is identified by a unique number. This number is forwarded to the reward function when it is called by the `NetLogoHeadlessActor`. The reward function calls `ql:get-group-list` with the identifying number and receives a list of `NLGroupChoices`. Besides the agents, an `NLGroupChoice` also contains a list of the agents' decisions. The agents and their choices are accessed by the reporters `ql:get-agents` and `ql:get-decisions`. The rewards are set to an `NLGroupChoice` by `ql:set-rewards`. The reward function can also be used to update the (NetLogo) agents directly, e.g. by moving the agents within the NetLogo world or by setting variables. Since the agents are passed from the main NetLogo instance, the changes take effect in this instance as well. Finally, the reward function must return a new list of `NLGroupChoices` that correspond to the received list but with the rewards attributes set.

[1]`https://groups.google.com/forum/#!msg/netlogo-devel/8oDmCRERDlQ/0IDZm0l5eNwJ`

C.4 The two-armed bandit problem

In figure C.3 and listing C.2, a second NetLogo model is given (downloadable as `NetLogo-ql/models/n-armed-bandit.nlogo`). It makes use of the parallel mode and implements the two-armed bandit problem. An agent, who is represented by a patch, chooses repeatedly between two alternatives. The reward of either choice is drawn from a normal distribution with mean `alt-1-reward` or `alt-2-reward` and standard deviation one. Different colors (white or grey) indicate the last decision of an agent.

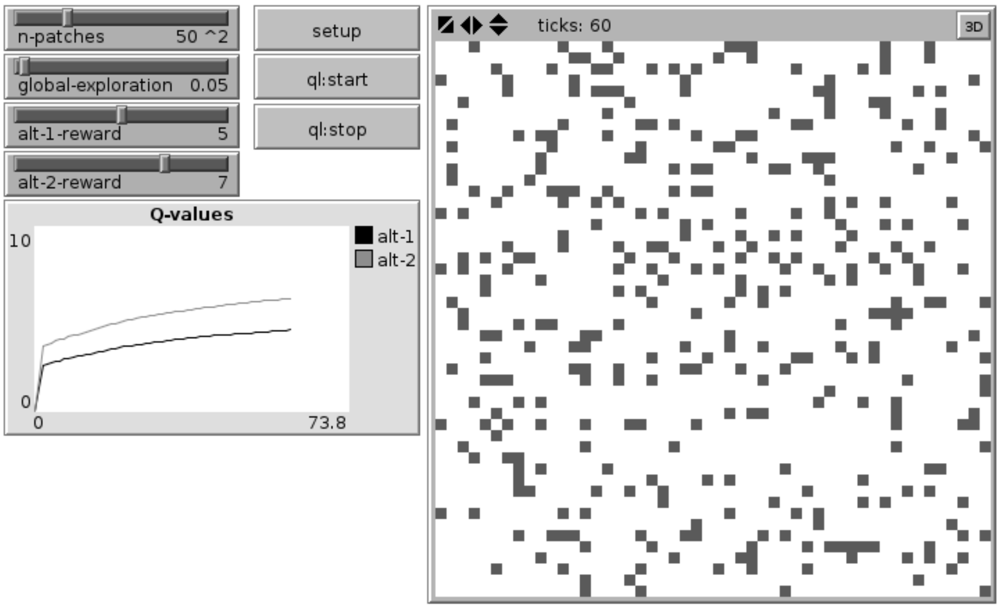


Figure C.3: NetLogo interface of the 2-armed bandit problem

The setup function of listing C.2 is similar to the one of the first example in section C.1. The `exploration-rate` and `exploration-method` are set for each agent, and the ql-extension is initialised by `ql:init`. Since there is no interaction between the agents, each group consists of a single agent. In line 13 of listing C.2, a list of groups is created, one group for each agent. At the same time, a list of alternatives is defined. The group structure is handed over to the extension in line 14. It is a static structure because the groups do not change during the simulation. In section C.7, it is demonstrated how to implement a dynamic group structure.

```
extensions[ql]
patches-own[ exploration-rate exploration-method q-values ]

to setup
  clear-all
  set-patch-size 400 / n-patches
  resize-world 0 (n-patches - 1) 0 (n-patches - 1)
  ask patches [
    set exploration-rate global-exploration
    set exploration-method "epsilon-greedy"
  ]
  ql:init patches
  let groups [ql:create-group (list (list self [0 1]))] of patches
  ql:set-group-structure groups
  reset-ticks
end

to-report get-rewards [ headless-id ]
  let group-list ql:get-group-list headless-id
  report map [reward ?] group-list
end

to-report reward [group]
  let agent first ql:get-agents group
  let decision first ql:get-decisions group
  ifelse decision = 0 [
    ask agent [set pcolor blue]
    report ql:set-rewards group (list random-normal alt-1-reward 1)
  ] [
    ask agent [set pcolor red]
    report ql:set-rewards group (list random-normal alt-2-reward 1)
  ]
end

to update
  tick
end
```

Listing C.2: NetLogo code of the 2-armed bandit problem

The simulation is started and stopped by `ql:start` and `ql:stop`. After starting the simulation, two functions are called repeatedly by the ql-extension and, hence, must be implemented in the NetLogo model. By default, the functions are named `get-rewards` and `update`. The names of the functions can be changed in the file `application.conf`. The first function is used to calculated the rewards of a group of agents. It comes with exactly one parameter (`headless-id`). The second function is executed repeatedly after every agent has received a reward. In listing C.2, a new tick is set, which updates the NetLogo interface.

The following list describes some commands of the ql-extension in detail:

- `ql:create-group` is a reporter that creates a group from a list of pairs. Each pair is a list with two elements: first, an agent and, second, a list of integers (the alternatives). An object of type `NLGroup` is returned.
- `ql:set-group-structure` takes a list of objects of type `NLGroup` as parameter. It sets a static group structure.
- `ql:start` or `ql:stop` starts or stops the simulation.
- `ql:get-group-list` can only be called from the reward function and must forward the `headless-id`. It returns a list of objects of type `NLGroupChoice`.
- `ql:get-agents` returns the list of NetLogo agents (turtles or patches) that are held by a `NLGroupChoice`.
- `ql:get-decisions` returns the list of decisions that are held by a `NLGroupChoice`. The indices of the decisions correspond to the indices of the agents that are held by the `NLGroupChoice` such that the decision at index i belongs to the agent at index i.
- `ql:set-rewards` sets a list of rewards for the decisions that are held by a `NLGroupChoice`. It returns a copy of the `NLGroupChoice` with the rewards attribute set. The indices of the rewards must correspond to the indices of the agents that are held by the `NLGroupChoice` such that the reward at index i belongs to the agent at index i.

C.5 Parameter sweeps

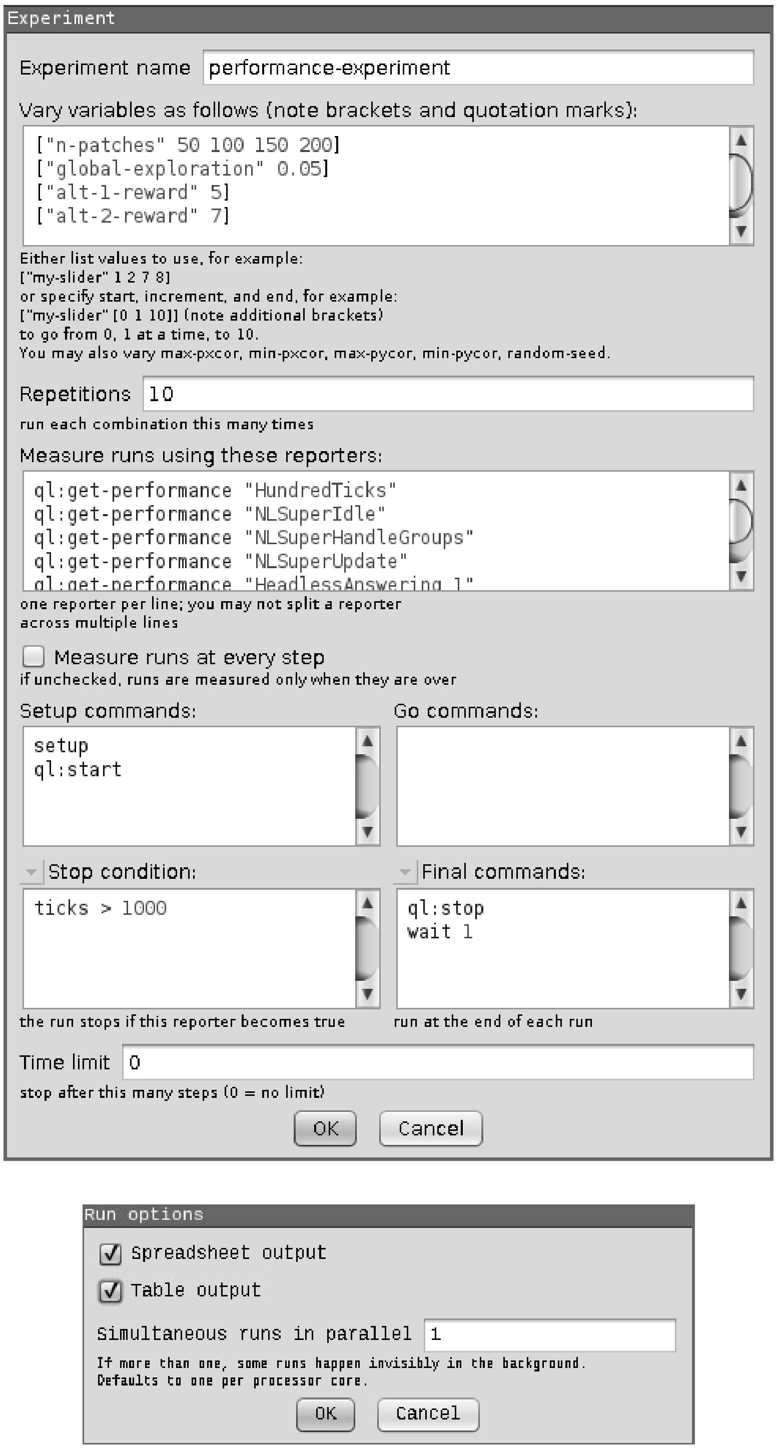
The ql-extension supports parameter sweeps with the BehaviourSpace of NetLogo (see `http://ccl.northwestern.edu/netlogo/docs/behaviorspace.html`). The setup slightly differs from the usual proceeding (see figure C.4). First, there are no “Go commands”. Instead, `ql:start` is called as “Setup command”. Second, `ql:stop` is added to “Final commands”. Since the BehaviourSpace recreates the agents instantly even if the ql-extension has not finished yet, an error occurs once in a while. By calling `wait 1` after `ql:stop`, this error is prevented. If the BehaviourSpace waits for one second, the ql-extension is usually ready for new agents. Third, a “Stop condition” must be present because the time limit does not work. Finally, only one experiment can run simultaneously (see lower window of figure C.4). The parallelisation is already built into the ql-extension.

This setup works as long as the measures are run at the end of the simulation. If the option “measure runs at every step” is enabled, a “Go command” that forces NetLogo to wait for the next tick must be added. For example:

```
to wait-for-tick
  set nextTick nextTick + 1
  while [ticks < nextTick] [
    random 100
    ; do useful stuff
    ; e.g. update globals here
  ]
end
```

It is also possible to run the experiments from command line (`http://ccl.northwestern.edu/netlogo/docs/behaviorspace.html#advanced`). As already stated, the number of threads is limited to one when using the ql-extension:

```
java -Xmx1024m -Dfile.encoding=UTF-8 -cp NetLogo.jar org.nlogo.
    headless.Main --model ql-model.nlogo --experiment performance-
    experiment --threads 1
```



Experiment name: performance-experiment

Vary variables as follows (note brackets and quotation marks):

```
["n-patches" 50 100 150 200]
["global-exploration" 0.05]
["alt-1-reward" 5]
["alt-2-reward" 7]
```

Either list values to use, for example:
["my-slider" 1 2 7 8]
or specify start, increment, and end, for example:
["my-slider" [0 1 10]] (note additional brackets)
to go from 0, 1 at a time, to 10.
You may also vary max-pxcor, min-pxcor, max-pycor, min-pycor, random-seed.

Repetitions: 10
run each combination this many times

Measure runs using these reporters:

```
ql:get-performance "HundredTicks"
ql:get-performance "NLSuperIdle"
ql:get-performance "NLSuperHandleGroups"
ql:get-performance "NLSuperUpdate"
ql:get-performance "HeadlessAnswering 1"
```

one reporter per line; you may not split a reporter
across multiple lines

☐ Measure runs at every step
if unchecked, runs are measured only when they are over

Setup commands:

```
setup
ql:start
```

Go commands:

Stop condition:

```
ticks > 1000
```

the run stops if this reporter becomes true

Final commands:

```
ql:stop
wait 1
```

run at the end of each run

Time limit: 0
stop after this many steps (0 = no limit)

✓ Spreadsheet output
✓ Table output

Simultaneous runs in parallel: 1
If more than one, some runs happen invisibly in the background.
Defaults to one per processor core.

Figure C.4: NetLogo interfaces of an experiment

C.6 Performance

An advantage of the ql-extension is the enhanced performance of the simulations, which is achieved by concurrency. The performance of a simulation can be measured by monitoring the `NetLogoSupervisor` and the `NetLogoHeadlessActors`. Different measures are included in the ql-extension and obtained by the reporter `ql:get-performance`. The reporter takes one of the following string parameters and returns the time in milliseconds:

- "HundredTicks" - the average time that is needed for 100 ticks.
- "NLSuperIdle" - the average time of the `NetLogoSupervisor` being idle, which means that it waits for the `NetLogoHeadlessActors` to finish the reward calculations.
- "NLSuperHandleGroups" - the average time of the `NetLogoSupervisor` forwarding the `NLGroups` to the `NetLogoHeadlessRouter` (this becomes relevant in case of a dynamic group structure because the primary NetLogo instance needs to be invoked).
- "NLSuperUpdate" - the average time of the `NetLogoSupervisor` executing the `update` procedure.
- "HeadlessIdle 1" - the average time of the first `NetLogoHeadlessActor` being idle, which means that it waits for the `NetLogoSupervisor` to forward `NLGroups` or for the NetLogo headless workspaces to calculate the rewards.
- "HeadlessHandleGroups 1" - the average time of the first `NetLogoHeadless-Actor` initiating the agents to make a decision.
- "HeadlessHandleChoices 1" - the average time of the first `NetLogoHeadless-Actor` handling the agents' decisions (calling the NetLogo headless workspace).
- "HeadlessAnswering 1" - the average time of the first NetLogo headless workspace waiting for the first `NetLogoHeadlessActor` to forward the list of `NLGroupChoices`.

By changing the number of the last four parameters, the corresponding values of the other `NetLogoHeadlessActors` are obtained. The usage of the performance measures must be enabled in the configuration file (`application.conf`).

In the following, it is shown that the number of `NetLogoHeadlessActors` is a relevant factor in regard to the performance of the simulation. A performance leak exists because the `NetLogoSupervisor` must wait for the `NetLogoHeadlessActors` to finish their calculations. This means that the performance of a simulation is optimised by increasing the number of concurrently working headless actors. With many of them, the `NetLogoSupervisor` is rarely idle, and the simulation runs faster. This should be evident by the performances measure "NLSuperIdle" and "HundredTicks". In case of the two-armed bandit simulation (section C.4), both measures are pictured in figure C.5 for different numbers of patches and different numbers of concurrently working `NetLogoHeadlessActors`.

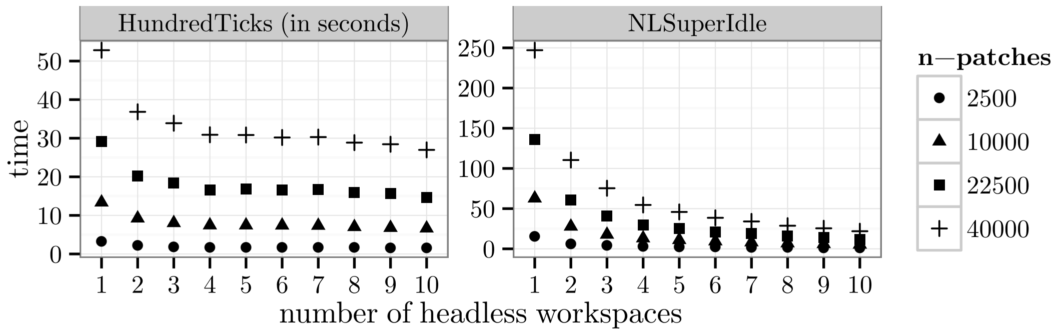


Figure C.5: Performance of the two-armed bandit simulation

Figure C.5 reveals that the performance of the simulation increases with the number of `NetLogoHeadlessActors`. Comparing the simulations with one and ten `NetLogoHeadlessActors`, the former needs twice as long as the latter for 100 ticks. Moreover, the second graph illustrates that there is not much room for improvement. The `NetLogoSupervisor` is almost never idle if ten `NetLogo-HeadlessActors` work concurrently.

Furthermore, the measures "NLSuperHandleGroups", "HeadlessHandleChoices", and "HeadlessAnswering" are very close to zero and independent of the number of patches or `NetLogoHeadlessActors`. The time that the `NetLogoSupervisor` needs to execute the `update` procedure ("NLSuperUpdate") increases with the number of patches. The remaining two measures ("HeadlessHandleGroups" and "HeadlessIdle") show similar developments as the total performance (see figure C.6). Since the list of `NLGroups` is distributed equally among the headless actors, the average waiting time as well as the average working time of the `NetLogo-HeadlessActors` decreases with their number.

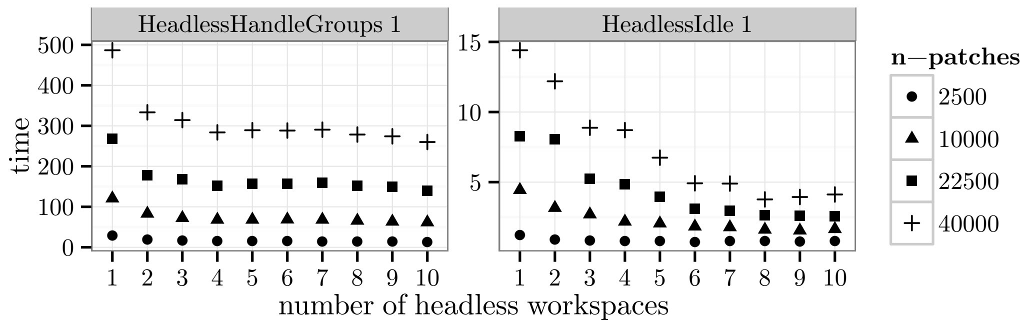


Figure C.6: Performance of the first `NetLogoHeadlessActor`

The optimal number of `NetLogoHeadlessActors` depends on the disposable computational resources. Every NetLogo headless workspace is started on a new thread. If ten `NetLogoHeadlessActors` are used, NetLogo occupies at least ten threads. These threads must block from time to time when waiting for the ql-extension. It is, therefore, beneficial to have at least the same number of threads available for the ql-extension. The minimum and maximum number of threads for the ql-extension are set in the configuration file (`application.conf`). Further experiments have shown that performance is best if the maximum number of threads is slightly above the number of `NetLogoHeadlessActors` (for example, maximal 12 threads if 10 headless actors are employed). The optimal number of threads and `NetLogoHeadlessActors` should be evaluated empirically given the maximum number of agents.

C.7 Dynamic group structures

In the simulations of section 7.1, two-person games were played with multiple partners. The simulations were run for 1 000 rounds. If the group structure had been fixed, agents with 2 partners would have had 2 000 choices and agents with 4 partners 4 000 choices during one simulation. Since the number of choices affects the outcome variables, such as Q-values and relative frequencies, simulations with different numbers of partners are not comparable.

One solution is that each agent has one choice per round regardless of the number of partners. In simulations with multiple partners, this requires a different group structure at each round. As stated in section C.3, the `NetLogoSupervisor` is responsible for updating the group structure. If `ql:set-group-structure` is not called after initialising the ql-extension, the `NetLogoSupervisor` executes the procedure `get-group` before every new round of decision-making. This procedure must, hence, be implemented in the NetLogo model (the name can be changed in the file `application.conf`). An example is given in listing C.3.

```
globals [ group-structure ]
turtles-own [ exploration-rate exploration-method ]

to setup
  clear-all
  create-turtles 100 [
    set exploration-rate 0.5
    set exploration-method "epsilon-greedy"
  ]
  ql:init turtles
  set group-structure []
  let i 0
  while [i < (count turtles)] [
    ask turtle i [
      let anotherTurtle turtle ((i + 1) mod (count turtles))
      let group ql:create-group (list
        (list self (n-values 2 [ ? ]))
        (list anotherTurtle (n-values 2 [ ? ])))
      set group-structure lput group group-structure
    ]
    set i i + 1
  ]
end

to-report get-groups
  report n-of 50 group-structure
end
```

Listing C.3: NetLogo code of a dynamic group structure

Similar to the “Small Worlds” model, which is available from the NetLogo commons (`http://ccl.northwestern.edu/netlogo/models/SmallWorlds`), the procedure of listing C.3 creates 100 turtles that are embedded in a perfect 1-lattice (see also section 7.1). After the global variable `group-structure` has been created, the command `ql:set-group-structure` is not called. Instead, the function `get-groups` is implemented.

The `NetLogoSupervisor` expects the function `get-groups` to return a list of `NLGroups`. These groups are either created in this function or in the setup. If created in the setup, `ql:init` must be called before the groups are created (see listing C.3). In the example, the function `get-groups` is used to randomly select 50 groups at each round.

In the present implementation, not every agent has exactly one choice at any given round. Some agents may have two choices, and some agents may have no choice. But, on average, an agent has one choice per round. A more accurate procedure is possible. But `get-groups` is one of the “performance bottlenecks” of the simulation. It should, therefore, be implemented as simple as possible.

Bibliography

Rauber, T. and G. Rünger (2013). *Parallel Programming* (2nd ed.). Springer.

Wyatt, D. (2013). *Akka concurrency.* artima.