overview

Plant Kingdom

February 16, 2014

Contents

1 Preliminaries

- Go through the PeerSim tutorial on how to create a cycle-based simulation in order to get a feel for how the environment operates.
- All needed libraries (including PeerSim) should already be in the `lib/` directory. The primary prerequisite is for Apache Ant, which is used for the build scripts. A relatively recent version of Java is also needed.
- To compile, simply run `ant compile` from the root of your branch. The unchecked conversion warnings can be safely ignored.

2 Running a Simulation

- A convenince script is used to launch the simulation. This script, `run`, is a Bourne shell script, and will only work on UNIX-like systems. If you are using Myconet on Windows, you will need to create an equivalent `run.bat` or `run.cmd` file by using this file as a guide for how to set the Java classpath.
- The configuration for an individual experiment is performed using a text file. All of the relevant parameters can be found in the `example-config.txt` file in the root of your branch.

Not all parameters are discussed; only those that you are most likely to be interested in changing are discussed.

Basic Parameters
`config.basename` A string to be used as the basis for output file names
`network.size` The number of peers/nodes in the simluation
`simulation.cycles` The number of rounds to execute the simulation
`random.seed` If uncommented, a particular random scenario can be reproduced

Myconet Parameters
`init.pow.max` Nodes are assigned capacities via a power-law distribution; this is the cutoff that determines the largest node in the system. It should be selected based on the simulation size. A value that is too high will result in a degenerate network with all nodes connected to a single superpeer; too small, and the network will be sprawling and inefficient. Consult the Myconet papers for suitable value for different network sizes.
`protocol.fng.strategy_map` The maps possible protocol states that a node can be in to a Strategy class that specifies the behavior to be followed. For ordinary Myconet this should be set to `fungus.BasicMyconetStrategyMap`, and for SODAP it should be set to `fungus.SemiBulwarkStrategyMap`.

You likely will not need to modify the core Myconet protocols, though if you are working on a SODAP-related project you will need to familiarize yourself with that part of the codebase.

Protocols
`protocol.dat fungus.HyphaData` A passive protocol serving as data storage. If you want to set a value for every node, this is the easiest place to store it. NOTE WELL: you cannot just set per-node values in a constructor, due to PeerSim's extensive use of clone()! To set a per-node value at the start of the simulation, you must use an initializer (see the classes associated with `init.*` entries).
`protocol.lnk fungus.HyphaLink` Stores the graph that defines the neighbor relationships, and provides methods to change those relationships.
`protocol.fng fungus.FungalGrowth` Consults the strategy map to determine what actions should be taken by a node, given its current state. Myconet protocol rules are implemented in each strategy.
`protocol.myc fungus.MycoCast` Used by nodes to find new candidate neighbors
`protocol.fail fungus.FailureAlerter` Only used by HITAP, but needed by all scenarios because it's not properly isolated from the rest of the codebase yet. Be sure that `inhibition_strategy_` and `bulwark_strategy` are both set to the appropriate null strategy.
`protocol.chem fungus.ChemicalManager` As with the failure alerter, only relevant to HITAP scenarios, but does need to be enabled.

Comment or uncomment a particular control to enable/disable it. The parameters for a control do not need to be commented out to disable it, as they will not be read if the control is not enabled.

Controls
`control.dnet* fungus.Nodulator` Use the Nodulator to define dynamics (attacks and churn). For complex scenarios, you may have to write your own control. If `add` is between -1 and 1, the number will be interpreted as a percentage of nodes (of the specified type) that should be killed. If outside this range, will be intepreted as a fixed number of nodes. `from` and `to` specify which rounds the Nodulator will operate, and `period` can be used to run every n-th cycle. The `type` can be set to select from different subsets of nodes: `all`, `hypha`, `immobile`, `branching`, `extending`, `biomass`, `largest`.
`control.lo fungus.LogObserver` Controls logging and log levels. Consult the config file for examples of how to use these settings. To just focus on particular classes, add `control.lo.classes.YourClass` lines.
`control.jv fungus.JungVisualizer` Enables a GUI for visualizations. This does not work well with many more than a few hundred nodes, and will need to be disabled for larger simulations. Also, note the the layout algorithm does not do a very good job of displaying SODAP networks once the biomass peers have begun to raise their parent targets.
`control.jv fungus.VisualizerTransformers` If you wish to tweak how the visualizer behaves, the easiest way is probably by subclassing VisualizerTransformers. The visualizer is written using JUNG, a Java graph library. Consult the JUNG docs for more details
`control.pcv fungus.PeerRatioChartFrame` An example of how to get live charts out of a simulation; this will need to be tweaked to your particular scenario.
`control.disconnect fungus.DisconnectControl` This is the control that makes SODAP work! If `control.disconnect.parent_strategy` is set to `fungus.NullParentStrategy`, then the simulation will behave as a normal Myconet scenario. If it is set to `fungus.TotalFailureParentStrategy`, then parent targets will be adjusted as per SODAP rules.
`control.protocols peersim.cdsim.FullNextCycle` This is kind of a weird one; this is defined so that the exact place where all of the per-node protocols should be run. This is used in `order.control`; see the discussion of ordering, below.

Order of execution is a significant consideration. The order of controls is specified by `order.control`, and the order of protocols by `order.protocols`. This is described in the PeerSim docs, and should be read carefully, as your simulation will be very difficult to understand and debug if you do not keep this information in mind.

- Controls are run **alphabetically**, unless `order.control` is specified. In that case, the specified, enabled controls run in that order, followed by all enabled controls that were not listed (in alphabetical order).
- Protocols follow similar rules. Specified, then alphabetical.

Recall, however, that controls run for the entire network, while protocols execute per node. In some cases, we want some controls to run, then let the protocols execute, then run the remaining controls. For that, the `control.protocols` line in the table above is used, which lets us specify `protocols` in the controls list. It is possible to intersperse controls and protocols further, but that gets more complex and you are unlikely to need it for this project.

This can be seen in the following line:

```
order.control shf dnet dnet2 disconnect protocols jg degree jv
```

In this case the `Shuffle` control is run first (to make sure that nodes are processed in a different order every cycle. Then the `dnet` Nodulators run, to kill nodes and add new ones. Following that the SODAP `disconnect` control adjusts parent targets. (This should more properly be implemented as a protocol, but was done as a control for speed, due to a looming deadline.) Then, all of the Myconet protocols are executed for the nodes. `jg` is the `JungGraphObserver` control, that creates an easy-to-manipulate graph data structure that can be used by other observers (so it is very important that it be run *after* all action is complete, but *before* the other observers that need it. `degree` calculates some graph statistics based on the JUNG graph, and `jv` is the visualizer, which is updated only after everything else is complete.

Annoyingly, a disabled control has to be removed from the order list, or it will result in an error. If you create a new control, remember that is will be run at the very end, unless you specify otherwise! This is probably not what you want.

3 Visualizer

Once you run the simulation with the visualizer enabled, you will see a number of buttons. Note that the position of the nodes on the screen does not represent their actual location. . . it is a layout algorith that attempts to make the neighbor relationships clear, which is the significant aspect. With the basic `VisualizerTransformers` enabled, small blue circles are biomass, red triangles are extending hyphae, yellow squares are branching hyphae, and green pentagons are immobile hyphae.

The JUNG layout code has a couple of bugs in it, which results in an occasional race condition and null pointer exception. Should this occur, just kill the simulation and restart.

- `freeze` pauses the layout algorithm, which can be useful when you are letting the simulation run for a number of rounds and don't care about what's happening in between. Clicking the button again restarts the layout engine.
- `capture` takes a picture of the displayed graph and saves it to the `capture/` directory using the basename specified in the config file.
- `pause` stops a running or walking simulation.
- `step` advances the simulation by one cycle
- `walk` starts the simulation advancing with a short delay between each cycle. If this delay is too short, you'll need to change it in the code.
- `run` starts the simulation running without pausing between cycles. If you click `run` (say, because you want to get to round 80, fast), it's best to freeze the layout to reduce the chance of the null pointer error.