NDlib Documentation

Release 5.0.0

Giulio Rossetti

Jul 07, 2020

Contents

NDlib is a Python software package that allows to describe, simulate, and study diffusion processes on complex networks.

Date	Python Versions	Main Author	GitHub	pypl
2020-07-07	3.x	Giulio Rossetti	Source	Distribution

If you use NDlib as support to your research consider citing:

> G. Rossetti, L. Milli, S. Rinzivillo, A. Sirbu, D. Pedreschi, F. Giannotti. **"NDlib: a Python Library to Model and Analyze Diffusion Processes Over Complex Networks"** Journal of Data Science and Analytics. 2017. DOI:0.1007/s41060-017-0086-6 (pre-print available on arXiv)
>
> G. Rossetti, L. Milli, S. Rinzivillo, A. Sirbu, D. Pedreschi, F. Giannotti. **"NDlib: Studying Network Diffusion Dynamics"** IEEE International Conference on Data Science and Advanced Analytics, DSAA. 2017.

CHAPTER 1

NDlib Dev Team

Name	Contribution
Giulio Rossetti	Library Design/Documentation
Letizia Milli	Epidemic Models
Alina Sirbu	Opinion Dynamics Model
Salvatore Rinzivillo	Visual Platform

1.1 Overview

`NDlib` is a Python language software package for the describing, simulate, and study diffusion processes on complex networks.

1.1.1 Who uses NDlib?

The potential audience for `NDlib` includes mathematicians, physicists, biologists, computer scientists, and social scientists.

1.1.2 Goals

`NDlib` is built upon the NetworkX python library and is intended to provide:

- tools for the study diffusion dynamics on social, biological, and infrastructure networks,
- a standard programming interface and diffusion models implementation that is suitable for many applications,
- a rapid development environment for collaborative, multidisciplinary, projects.

1.1.3 The Python NDlib library

`NDlib` is a powerful Python package that allows simple and flexible simulations of networks diffusion processes.

Most importantly, `NDlib`, as well as the Python programming language, is free, well-supported, and a joy to use.

1.1.4 Free software

`NDlib` is free software; you can redistribute it and/or modify it under the terms of the BSD License. We welcome contributions from the community.

1.1.5 EU H2020

`NDlib` is a result of two European H2020 projects:

- CIMPLEX "Bringing CItizens, Models and Data together in Participatory, Interactive SociaL EXploratories": under the funding scheme "FETPROACT-1-2014: Global Systems Science (GSS)", grant agreement #641191.
- SoBigData "Social Mining & Big Data Ecosystem": under the scheme "INFRAIA-1-2014-2015: Research Infrastructures", grant agreement #654024.

1.2 Download

1.2.1 Software

Source and binary releases: https://pypi.python.org/pypi/ndlib

Github (latest development): https://github.com/GiulioRossetti/ndlib

Github NDlib-REST: https://github.com/GiulioRossetti/ndlib-rest

Github NDlib-Viz: https://github.com/rinziv/NDLib-Viz

1.2.2 Documentation

1.3 Installing NDlib

Before installing `NDlib`, you need to have setuptools installed.

1.3.1 Quick install

Get `NDlib` from the Python Package Index at pypl.

or install it with

```
pip install ndlib
```

and an attempt will be made to find and install an appropriate version that matches your operating system and Python version.

You can install the development version with

```
pip install git+http://github.com/GiulioRossetti/ndlib.git
```

1.3.2 Installing from source

You can install from source by downloading a source archive file (tar.gz or zip) or by checking out the source files from the GitHub source code repository.

`NDlib` is a pure Python package; you don't need a compiler to build or install it.

Source archive file

Download the source (tar.gz or zip file) from pypl or get the latest development version from GitHub

Unpack and change directory to the source directory (it should have the files README.txt and setup.py).

Run python setup.py install to build and install

GitHub

Clone the NDlib repostitory (see GitHub for options)

```
git clone https://github.com/GiulioRossetti/ndlib.git
```

Change directory to ndlib

Run python setup.py install to build and install

If you don't have permission to install software on your system, you can install into another directory using the –user, –prefix, or –home flags to setup.py.

For example

```
python setup.py install --prefix=/home/username/python
```

or

```
python setup.py install --home=~
```

or

```
python setup.py install --user
```

If you didn't install in the standard Python site-packages directory you will need to set your PYTHONPATH variable to the alternate location. See http://docs.python.org/2/install/index.html#search-path for further details.

1.3.3 Requirements

Python

To use NDlib you need Python 2.7, 3.2 or later.

The easiest way to get Python and most optional packages is to install the Enthought Python distribution "Canopy" or using Anaconda.

There are several other distributions that contain the key packages you need for scientific computing.

Required packages

The following are packages required by NDlib.

NetworkX

Provides the graph representation used by the diffusion models implemented in NDlib.

Download: http://networkx.github.io/download.html

Optional packages

The following are optional packages that NDlib can use to provide additional functions.

Bokeh

Provides support to the visualization facilities offered by NDlib.

Download: http://bokeh.pydata.org/en/latest/

Other packages

These are extra packages you may consider using with NDlib

IPython, interactive Python shell, http://ipython.scipy.org/

1.4 Tutorial

NDlib is built upon networkx and is designed to configure, simulate and visualize diffusion experiments.

Here you can find a few examples to get started with ndlib: for a more comprehensive tutorial check the official Jupyter Notebook.

1.4.1 Installation

In order to install the latest version of the library (with visualization facilities) use

```
pip install ndlib
```

1.4.2 Chose a Diffusion model

Let's start importing the required libraries

```
import networkx as nx
import ndlib.models.epidemics as ep
```

Once imported the epidemic model module and the networkx library we can initialize the simulation:

```
# Network Definition
g = nx.erdos_renyi_graph(1000, 0.1)

# Model Selection
model = ep.SIRModel(g)
```

1.4.3 Configure the simulation

Each model has its own parameters: in order to completely instantiate the simulation we need to specify them using a `Configuration` object:

```
import ndlib.models.ModelConfig as mc

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('beta', 0.001)
config.add_model_parameter('gamma', 0.01)
config.add_model_parameter("fraction_infected", 0.05)
model.set_initial_status(config)
```

The model configuration allows to specify model parameters (as in this scenario) as well as nodes' and edges' ones (e.g. individual thresholds).

Moreover it allows to specify the initial fraction of infected nodes using the `fraction_infected` model parameter.

It is also possible to explicitly specify an initial set of infected nodes: see *ModelConfig* for the complete set of use cases.

1.4.4 Execute the simulation

In order to execute the simulation one, or more, iterations must be required using the `model.iteration()` and/or `model.iteration_bunch(n_iterations)` methods.

```
# Simulation
iterations = model.iteration_bunch(200)
trends = model.build_trends(iterations)
```

1.4.5 Visualize the results

At the end of the simulation the diffusion trend can be visualized as follows (for `matplotlib` change `ndlib.viz.bokeh` in `ndlib.viz.mpl`)

```
from bokeh.io import output_notebook, show
from ndlib.viz.bokeh.DiffusionTrend import DiffusionTrend

viz = DiffusionTrend(model, trends)
p = viz.plot(width=400, height=400)
show(p)
```

Furthermore, a prevalence plot is also made available.

The prevalence plot captures the variation (delta) of nodes for each status in consecutive iterations.

```
from ndlib.viz.bokeh.DiffusionPrevalence import DiffusionPrevalence

viz2 = DiffusionPrevalence(model, trends)
p2 = viz2.plot(width=400, height=400)
show(p2)
```

Multiple plots can be combined in a multiplot to provide a complete description of the diffusive process

```
from ndlib.viz.bokeh.MultiPlot import MultiPlot
vm = MultiPlot()
vm.add_plot(p)
vm.add_plot(p2)
m = vm.plot()
show(m)
```

Multiplots - implemented only for the `bokeh` provider - are also useful to compare different diffusion models applied to the same graph (as well as a same model instantiated with different parameters)

```
import ndlib.models.epidemics as ep

vm = MultiPlot()
vm.add_plot(p)

# SIS
sis_model = ep.SISModel(g)
config = mc.Configuration()
config.add_model_parameter('beta', 0.001)
config.add_model_parameter('lambda', 0.01)
config.add_model_parameter("fraction_infected", 0.05)
sis_model.set_initial_status(config)
iterations = sis_model.iteration_bunch(200)
trends = sis_model.build_trends(iterations)

viz = DiffusionTrend(sis_model, trends)
p3 = viz.plot(width=400, height=400)
vm.add_plot(p3)

# SI
si_model = ep.SIModel(g)
config = mc.Configuration()
config.add_model_parameter('beta', 0.001)
config.add_model_parameter("fraction_infected", 0.05)
si_model.set_initial_status(config)
iterations = si_model.iteration_bunch(200)
trends = si_model.build_trends(iterations)

viz = DiffusionTrend(si_model, trends)
p4 = viz.plot(width=400, height=400)
vm.add_plot(p4)

# Threshold
th_model = ep.ThresholdModel(g)
config = mc.Configuration()

# Set individual node threshold
threshold = 0.40
for n in g.nodes():
```

(continues on next page)

(continued from previous page)

```
        config.add_node_configuration("threshold", n, threshold)

config.add_model_parameter("fraction_infected", 0.30)
th_model.set_initial_status(config)
iterations = th_model.iteration_bunch(60)
trends = th_model.build_trends(iterations)

viz = DiffusionTrend(th_model, trends)
p5 = viz.plot(width=400, height=400)
vm.add_plot(p5)

m = vm.plot()
show(m)
```

1.5 Network Diffusion Library Reference

In this section are introduced the components that constitute `NDlib`, namely

- The implemented diffusion models (organized in **Epidemics** and **Opinion Dynamics**)
- The methodology adopted to configure a general simulation
- The visualization facilities embedded in the library to explore the results

Advanced topics (Custom model definition, Network Diffusion Query language (NDQL), Experiment Server and Visual Framework) are reported in separate sections.

1.5.1 Diffusion Models

The analysis of diffusive phenomena that unfold on top of complex networks is a task able to attract growing interests from multiple fields of research.

In order to provide a succinct framing of such complex and extensively studied problem it is possible to split the related literature into two broad, related, sub-classes: **Epidemics** and **Opinion Dynamics**.

Moreover, `NDlib` also supports the simulation of diffusive processes on top of evolving network topologies: the **Dynamic Network Models** section the ones `NDlib` implements.

Epidemics

When we talk about epidemics, we think about contagious diseases caused by biological pathogens, like influenza, measles, chickenpox and sexually transmitted viruses that spread from person to person. However, other phenomena can be linked to the concept of epidemic: think about the spread of computer virus[1] where the agent is a malware that can transmit a copy of itself from computer to computer, or the spread of mobile phone virus[2][3], or the diffusion

[1] P. Szor, "Fighting computer virus attacks." USENIX, 2004.

[2] S. Havlin, "Phone infections," Science, 2009.

[3] P.Wang,M.C.Gonzalez,R.Menezes,andA.L.Baraba si,"Understanding the spread of malicious mobile-phone programs and their damage potential," International Journal of Information Security, 2013.

of knowledge, innovations, products in an online social network[4] - the so-called "social contagion", where people are making decision to adopt a new idea or innovation.

Several elements determine the patterns by which epidemics spread through groups of people: the properties carried by the pathogen (its contagiousness, the length of its infectious period and its severity), the structure of the network as well as the mobility patterns of the people involved. Although often treated as similar processes, diffusion of information and epidemic spreading can be easily distinguished by a single feature: the degree of activeness of the subjects they affect.

Indeed, the spreading process of a virus does not require an active participation of the people that catch it (i.e., even though some behaviors acts as contagion facilitators – scarce hygiene, moist and crowded environment – we can assume that no one chooses to get the flu on purpose); conversely, we can argue that the diffusion of an idea, an innovation, or a trend strictly depend not only by the social pressure but also by individual choices.

In `NDlib` are implemented the following **Epidemic** models:

SI

The SI model was introduced in 1927 by Kermack[1].

In this model, during the course of an epidemics, a node is allowed to change its status only from **Susceptible** (S) to **Infected** (I).

The model is instantiated on a graph having a non-empty set of infected nodes.

SI assumes that if, during a generic iteration, a susceptible node comes into contact with an infected one, it becomes infected with probability β: once a node becomes infected, it stays infected (the only transition allowed is S→I).

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
beta	Model	float in [0, 1]		True	Infection probability

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

[4] R. S. Burt, "Social Contagion and Innovation: Cohesion Versus Structural Equivalence," American Journal of Sociology, 1987.

[1] W. O. Kermack and A. McKendrick, "A Contribution to the Mathematical Theory of Epidemics," Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, vol. 115, no. 772, pp. 700–721, Aug. 1927.

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.SIModel.**SIModel**(*graph*, *seed=None*)
Model Parameters to be specified via ModelConfig

Parameters **beta** – The infection rate (float value in [0,1])

SIModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

SIModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

SIModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

SIModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

SIModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

SIModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

SIModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of an SI simulation on a random graph: we set the initial set of infected nodes as 5% of the overall population and a probability of infection of 1%.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SIModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.01)
cfg.add_model_parameter("fraction_infected", 0.05)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
```

SIS

The SIS model was introduced in 1927 by Kermack[1].

In this model, during the course of an epidemics, a node is allowed to change its status from **Susceptible** (S) to **Infected** (I).

The model is instantiated on a graph having a non-empty set of infected nodes.

SIS assumes that if, during a generic iteration, a susceptible node comes into contact with an infected one, it becomes infected with probability beta, than it can be switch again to susceptible with probability lambda (the only transition allowed are S→I→S).

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

[1] W. O. Kermack and A. McKendrick, "A Contribution to the Mathematical Theory of Epidemics," Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, vol. 115, no. 772, pp. 700–721, Aug. 1927

Parameters

Name	Type	Value Type	Default	Mandatory	Description
beta	Model	float in [0, 1]		True	Infection probability
lambda	Model	float in [0, 1]		True	Recovery probability

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.SISModel.**SISModel**(*graph*, *seed=None*)
Model Parameters to be specified via ModelConfig

Parameters

- **beta** – The infection rate (float value in [0,1])
- **lambda** – The recovery rate (float value in [0,1])

SISModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

SISModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

SISModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

SISModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

SISModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

SISModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

SISModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of an SIS simulation on a random graph: we set the initial set of infected nodes as 5% of the overall population, a probability of infection of 1%, and a probability of recovery of 0.5%.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SISModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.01)
cfg.add_model_parameter('lambda', 0.005)
cfg.add_model_parameter("fraction_infected", 0.05)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
```

SIR

The SIR model was introduced in 1927 by Kermack[1].

In this model, during the course of an epidemics, a node is allowed to change its status from **Susceptible** (S) to **Infected** (I), then to **Removed** (R).

The model is instantiated on a graph having a non-empty set of infected nodes.

[1] W. O. Kermack and A. McKendrick, "A Contribution to the Mathematical Theory of Epidemics," Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, vol. 115, no. 772, pp. 700–721, Aug. 1927

SIR assumes that if, during a generic iteration, a susceptible node comes into contact with an infected one, it becomes infected with probability beta, than it can be switch to removed with probability gamma (the only transition allowed are S→I→R).

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1
Removed	2

Parameters

Name	Type	Value Type	Default	Mandatory	Description
beta	Model	float in [0, 1]		True	Infection probability
gamma	Model	float in [0, 1]		True	Removal probability

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.SIRModel.**SIRModel**(*graph*, *seed=None*)
Model Parameters to be specified via ModelConfig

Parameters

- **beta** – The infection rate (float value in [0,1])
- **gamma** – The recovery rate (float value in [0,1])

SIRModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

SIRModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

SIRModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

SIRModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

SIRModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

SIRModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

SIRModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of an SIR simulation on a random graph: we set the initial set of infected nodes as 5% of the overall population, a probability of infection of 1%, and a removal probability of 0.5%.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.01)
cfg.add_model_parameter('gamma', 0.005)
cfg.add_model_parameter("fraction_infected", 0.05)
```

(continues on next page)

(continued from previous page)

```
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
```

SEIR

In the SEIR model[1], during the course of an epidemics, a node is allowed to change its status from **Susceptible** (S) to **Exposed** (E) to **Infected** (I), then to **Removed** (R).

The model is instantiated on a graph having a non-empty set of infected nodes.

SEIR assumes that if, during a generic iteration, a susceptible node comes into contact with an infected one, it becomes infected after an exposition period with probability beta, than it can switch to removed with probability gamma (the only transition allowed are S→E→I→R).

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1
Exposed	2
Removed	3

Parameters

Name	Type	Value Type	Default	Mandatory	Description
beta	Model	float in [0, 1]		True	Infection probability
gamma	Model	float in [0, 1]		True	Removal probability
alpha	Model	float in [0, 1]		True	Latent period

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

[1] J.L. Aron and I.B. Schwartz. Seasonality and period-doubling bifurcations in an epidemic model. Journal Theoretical Biology, 110:665-679, 1984

Configure

class ndlib.models.epidemics.SEIRModel.**SEIRModel**(*graph*, *seed=None*)

SEIRModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

SEIRModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

SEIRModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

SEIRModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

SEIRModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

SEIRModel.**iteration**(*self*)
Execute a single model iteration

Parameters **node_status** – if the incremental node status has to be returned.

Returns Iteration_id, (optional) Incremental node status (dictionary node->status), Status count (dictionary status->node count), Status delta (dictionary status->node delta)

SEIRModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of an SEIR simulation on a random graph: we set the initial set of infected nodes as % of the overall population, a probability of infection of 1%, a removal probability of 0.5% and an incubation period of 5% (e.g. 20 iterations).

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SEIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.01)
cfg.add_model_parameter('gamma', 0.005)
cfg.add_model_parameter('alpha', 0.05)
cfg.add_model_parameter("fraction_infected", 0.05)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
```

SEIS

In the SEIS model, during the course of an epidemics, a node is allowed to change its status from **Susceptible** (S) to **Exposed** (E) to **Infected** (I), then again to **Susceptible** (S).

The model is instantiated on a graph having a non-empty set of infected nodes.

SEIS assumes that if, during a generic iteration, a susceptible node comes into contact with an infected one, it becomes infected after an exposition period with probability beta, than it can switch back to susceptible with probability lambda (the only transition allowed are S→E→I→S).

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1
Exposed	2

Parameters

Name	Type	Value Type	Default	Mandatory	Description
beta	Model	float in [0, 1]		True	Infection probability
lambda	Model	float in [0, 1]		True	Removal probability
alpha	Model	float in [0, 1]		True	Latent period

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]

- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.SEISModel.**SEISModel**(*graph*, *seed=None*)
Model Parameters to be specified via ModelConfig

Parameters

- **beta** – The infection rate (float value in [0,1])
- **lambda** – The recovery rate (float value in [0,1])

SEISModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

SEISModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

SEISModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

SEISModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

SEISModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

SEISModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

SEISModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of an SEIS simulation on a random graph: we set the initial set of infected nodes as 5% of the overall population, a probability of infection of 1%, a removal probability of 0.5% and an latent period of 5% (e.g. 20 iterations).

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SEISModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.01)
cfg.add_model_parameter('lambda', 0.005)
cfg.add_model_parameter('alpha', 0.05)
cfg.add_model_parameter("fraction_infected", 0.05)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
```

SWIR

The SWIR model was introduced in 2017 by Lee et al.[1].

In this model, during the epidemics, a node is allowed to change its status from **Susceptible** (S) to **Weakened** (W) or **Infected** (I), then to **Removed** (R).

The model is instantiated on a graph having a non-empty set of infected nodes.

At time *t* a node in the state I is selected randomly and the states of all neighbors are checked one by one. If the state of a neighbor is S then this state changes either i) to I with probability *kappa* or ii) to W with probability *mu*. If the state of a neighbor is W then the state W changes to I with probability *nu*. We repeat the above process for all nodes in state I and then changes to R for each associated node.

Statuses

During the simulation a node can experience the following statuses:

[1] D. Lee, W. Choi, J. Kertész, B. Kahng. "Universal mechanism for hybrid percolation transitions". Scientific Reports, vol. 7(1), 5723, 2017.

Name	Code
Susceptible	0
Infected	1
Weakened	2
Removed	3

Parameters

Name	Type	Value Type	Default	Mandatory	Description
kappa	Model	float in [0, 1]		True	
mu	Model	float in [0, 1]		True	
nu	Model	float in [0, 1]		True	

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.SWIRModel.**SWIRModel**(*graph*, *seed=None*)

SWIRModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

SWIRModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

SWIRModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

SWIRModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

SWIRModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

SWIRModel.**iteration**(*self*)
Execute a single model iteration

Parameters **node_status** – if the incremental node status has to be returned.

Returns Iteration_id, (optional) Incremental node status (dictionary node->status), Status count (dictionary status->node count), Status delta (dictionary status->node delta)

SWIRModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of an SEIR simulation on a random graph: we set the initial set of infected nodes as % of the overall population, a probability of infection of 1%, a removal probability of 0.5% and an latent period of 5% (e.g. 20 iterations).

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SWIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('kappa', 0.01)
cfg.add_model_parameter('mu', 0.005)
cfg.add_model_parameter('nu', 0.05)
cfg.add_model_parameter("fraction_infected", 0.05)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Threshold

The Threshold model was introduced in 1978 by Granovetter[1].

[1]

In this model during an epidemics, a node has two distinct and mutually exclusive behavioral alternatives, e.g., the decision to do or not do something, to participate or not participate in a riot.

Node's individual decision depends on the percentage of its neighbors have made the same choice, thus imposing a threshold.

The model works as follows: - each node has its own threshold; - during a generic iteration every node is observed: iff the percentage of its infected neighbors is grater than its threshold it becomes infected as well.

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
threshold	Node	float in [0, 1]	0.1	False	Individual threshold

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.ThresholdModel.**ThresholdModel**(*graph*, *seed=None*)

Node Parameters to be specified via ModelConfig

Parameters **threshold** – The node threshold. If not specified otherwise a value of 0.1 is assumed for all nodes.

ThresholdModel.**__init__**(*graph*)

Model Constructor

Parameters **graph** – A networkx graph object

ThresholdModel.**set_initial_status**(*self*, *configuration*)

Set the initial model configuration

M. Granovetter, "Threshold models of collective behavior," The American Journal of Sociology, vol. 83, no. 6, pp. 1420–1443, 1978.

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

ThresholdModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

ThresholdModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

ThresholdModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

ThresholdModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

ThresholdModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Threshold model simulation on a random graph: we set the initial set of infected nodes as 1% of the overall population, and assign a threshold of 0.25 to all the nodes.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.ThresholdModel(g)

# Model Configuration
config = mc.Configuration()
```

(continues on next page)

(continued from previous page)

```
config.add_model_parameter('fraction_infected', 0.1)

# Setting node parameters
threshold = 0.25
for i in g.nodes():
    config.add_node_configuration("threshold", i, threshold)

model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Generalised Threshold

The Generalised Threshold model was introduced in 2017 by Török and Kertesz[1].

In this model, during an epidemics, a node is allowed to change its status from **Susceptible** to **Infected**.

The model is instantiated on a graph having a non-empty set of infected nodes.

The model is defined as follows:

1. At time *t* nodes become Infected with rate *mu t/tau*
2. Nodes for which the ratio of the active friends dropped below the threshold are moved to the Infected queue
3. Nodes in the Infected queue become infected with rate *tau*. If this happens check all its friends for threshold

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
threshold	Node	float in [0, 1]	0.1	False	Individual threshold
tau	Model	int		True	Adoption threshold rate
mu	Model	int		True	Exogenous timescale

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

[1] János Török and János Kertész "Cascading collapse of online social networks" Scientific reports, vol. 7 no. 1, 2017

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.GeneralisedThresholdModel.**GeneralisedThresholdModel**(*graph*, *seed=None*)

Node Parameters to be specified via ModelConfig

Parameters **threshold** – The node threshold. If not specified otherwise a value of 0.1 is assumed for all nodes.

GeneralisedThresholdModel.**__init__**(*graph*)
Model Constructor :param graph: A networkx graph object

GeneralisedThresholdModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

GeneralisedThresholdModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

GeneralisedThresholdModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

GeneralisedThresholdModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

GeneralisedThresholdModel.**iteration**(*self*)
Execute a single model iteration :return: Iteration_id, Incremental node status (dictionary node->status)

GeneralisedThresholdModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Threshold model simulation on a random graph: we set the initial set of infected nodes as 1% of the overall population, and assign a threshold of 0.25 to all the nodes.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.GeneralisedThresholdModel(g)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)
config.add_model_parameter('tau', 5)
config.add_model_parameter('mu', 5)

# Setting node parameters
threshold = 0.25
for i in g.nodes():
    config.add_node_configuration("threshold", i, threshold)

model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Kertesz Threshold

The Kertesz Threshold model was introduced in 2015 by Ruan et al.[1] and it is an extension of the Watts threshold model[2].

The authors extend the classical model introducing a density **r** of blocked nodes – nodes which are immune to social influence – and a probability of spontaneous adoption **p** to capture external influence.

Thus, the model distinguishes three kinds of node: **Blocked** (B), **Susceptible** (S) and **Adoptiong** (A). The latter class breaks into two categories: vulnerable and stable nodes. A node can adopt either under its neighbors' influence, or spontaneously, due to endogenous effects.

Statuses

During the simulation a node can experience the following statuses:

[1] 26. Ruan, G. In iguez, M. Karsai, and J. Kertesz, "Kinetics of social contagion," Phys. Rev. Lett., vol. 115, p. 218702, Nov 2015.

[2] D. J. Watts, "A simple model of global cascades on random networks," Proceedings of the National Academy of Sciences, vol. 99, no. 9, pp. 5766–5771, 2002.

Name	Code
Susceptible	0
Infected	1
Blocked	-1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
adopter_rate	Model	float in [0, 1]	0	False	Exogenous adoption rate
percentage_blocked	Model	float in [0, 1]	0.1	False	Blocked nodes
threshold	Node	float in [0, 1]	0.1	False	Individual threshold

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The initial blocked nodes can be defined via:

- **percentage_blocked**: Model Parameter, float in [0, 1]
- **Blocked**: Status Parameter, set of nodes

In both cases, the two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.KerteszThresholdModel.**KerteszThresholdModel**(*graph*, *seed=None*)

Node/Model Parameters to be specified via ModelConfig

Parameters

- **threshold** – The node threshold. As default a value of 0.1 is assumed for all nodes.
- **adopter_rate** – The probability of spontaneous adoptions. Defaults value 0.
- **fraction_infected** – The percentage of blocked nodes. Default value 0.1.

KerteszThresholdModel.**__init__**(*graph*)

Model Constructor

Parameters **graph** – A networkx graph object

KerteszThresholdModel.**set_initial_status**(*self*, *configuration*)

Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

KerteszThresholdModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

KerteszThresholdModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

KerteszThresholdModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

KerteszThresholdModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

KerteszThresholdModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Kertesz Threshold model simulation on a random graph: we set the initial infected as well blocked node sets equals to the 10% of the overall population, assign a threshold of 0.25 to all the nodes and impose an probability of spontaneous adoptions of 40%.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.KerteszThresholdModel(g)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('adopter_rate', 0.4)
config.add_model_parameter('percentage_blocked', 0.1)
config.add_model_parameter('fraction_infected', 0.1)
```

(continues on next page)

(continued from previous page)

```
# Setting node parameters
threshold = 0.25
for i in g.nodes():
    config.add_node_configuration("threshold", i, threshold)

model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Independent Cascades

The Independent Cascades model was introduced by Kempe et all in 2003[1].

This model starts with an initial set of **active** nodes A0: the diffusive process unfolds in discrete steps according to the following randomized rule:

- When node v becomes active in step t, it is given a single chance to activate each currently inactive neighbor w; it succeeds with a probability p(v,w).
- If w has multiple newly activated neighbors, their attempts are sequenced in an arbitrary order.
- If v succeeds, then w will become active in step t + 1; but whether or not v succeeds, it cannot make any further attempts to activate w in subsequent rounds.
- The process runs until no more activations are possible.

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1
Removed	2

Parameters

Name	Type	Value Type	Default	Mandatory	Description
Edge threshold	Edge	float in [0, 1]	0.1	False	Edge threshold

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

[1] D. Kempe, J. Kleinberg, and E. Tardos, "Maximizing the spread of influence through a social network," in Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ser. KDD '03, 2003, pp. 137–146.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.IndependentCascadesModel.**IndependentCascadesModel**(*graph*, *seed=None*)

Edge Parameters to be specified via ModelConfig

Parameters **threshold** – The edge threshold. As default a value of 0.1 is assumed for all edges.

IndependentCascadesModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

IndependentCascadesModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

IndependentCascadesModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

IndependentCascadesModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

IndependentCascadesModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

IndependentCascadesModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

IndependentCascadesModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of an Independent Cascades model simulation on a random graph: we set the initial set of infected nodes as 1% of the overall population, and assign a threshold of 0.1 to all the edges.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.IndependentCascadesModel(g)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Setting the edge parameters
threshold = 0.1
for e in g.edges():
    config.add_edge_configuration("threshold", e, threshold)

model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Profile

The Profile model was introduced in 2017 by Milli et al.[1].

The Profile model assumes that the diffusion process is only apparent; each node decides to adopt or not a given behavior – once known its existence – only on the basis of its own interests.

In this scenario the peer pressure is completely ruled out from the overall model: it is not important how many of its neighbors have adopted a specific behaviour, if the node does not like it, it will not change its interests.

Each node has its own profile describing how many it is likely to accept a behaviour similar to the one that is currently spreading.

The diffusion process starts from a set of nodes that have already adopted a given behaviour S:

- for each of the susceptible nodes' in the neighborhood of a node u in S an unbalanced coin is flipped, the unbalance given by the personal profile of the susceptible node;
- if a positive result is obtained the susceptible node will adopt the behaviour, thus becoming infected.
- if the **blocked** status is enabled, after having rejected the adoption with probability `blocked` a node becomes immune to the infection.
- every iteration `adopter_rate` percentage of nodes spontaneous became infected to endogenous effects.

[1] Letizia Milli, Giulio Rossetti, Dino Pedreschi, Fosca Giannotti, "Information Diffusion in Complex Networks: The Active/Passive Conundrum," Proceedings of International Conference on Complex Networks and their Applications, (pp. 305-313). Springer, Cham. 2017

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1
Blocked	-1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
profile	Node	float in [0, 1]	0.1	False	Node profile
blocked	Model	float in [0, 1]	0	False	Blocked nodes
adopter_rate	Model	float in [0, 1]	0	False	Autonomous adoption

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.ProfileModel.**ProfileModel**(*graph*, *seed=None*)

Node Parameters to be specified via ModelConfig

Parameters **profile** – The node profile. As default a value of 0.1 is assumed for all nodes.

ProfileModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

ProfileModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

ProfileModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

ProfileModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

ProfileModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

ProfileModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

ProfileModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Profile model simulation on a random graph: we set the initial infected node set to the 10% of the overall population and assign a profile of 0.25 to all the nodes.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.ProfileModel(g)
config = mc.Configuration()
config.add_model_parameter('blocked', 0)
config.add_model_parameter('adopter_rate', 0)
config.add_model_parameter('fraction_infected', 0.1)

# Setting nodes parameters
profile = 0.15
for i in g.nodes():
    config.add_node_configuration("profile", i, profile)

model.set_initial_status(config)
```

(continues on next page)

(continued from previous page)

```
# Simulation execution
iterations = model.iteration_bunch(200)
```

Profile Threshold

The Profile-Threshold model was introduced in 2017 by Milli et al.[1].

The Profile-Threshold model assumes the existence of node profiles that act as preferential schemas for individual tastes but relax the constraints imposed by the Profile model by letting nodes influenceable via peer pressure mechanisms.

The peer pressure is modeled with a threshold.

The diffusion process starts from a set of nodes that have already adopted a given behaviour S:

- for each of the susceptible node an unbalanced coin is flipped if the percentage of its neighbors that are already infected excedes its threhosld. As in the Profile Model the coin unbalance is given by the personal profile of the susceptible node;
- if a positive result is obtained the susceptible node will adopt the behaviour, thus becoming infected.
- if the **blocked** status is enabled, after having rejected the adoption with probability `blocked` a node becomes immune to the infection.
- every iteration `adopter_rate` percentage of nodes spontaneous became infected to endogenous effects.

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1
Blocked	-1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
threshold	Node	float in [0, 1]	0.1	False	Individual threshold
profile	Node	float in [0, 1]	0.1	False	Node profile
blocked	Model	float in [0, 1]	0	False	Blocked nodes
adopter_rate	Model	float in [0, 1]	0	False	Autonomous adoption

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

[1] Letizia Milli, Giulio Rossetti, Dino Pedreschi, Fosca Giannotti, "Information Diffusion in Complex Networks: The Active/Passive Conundrum," Proceedings of International Conference on Complex Networks and their Applications, (pp. 305-313). Springer, Cham. 2017

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.epidemics.ProfileThresholdModel.**ProfileThresholdModel**(*graph*, *seed=None*)

Node Parameters to be specified via ModelConfig

Parameters

- **profile** – The node profile. As default a value of 0.1 is assumed for all nodes.
- **threshold** – The node threshold. As default a value of 0.1 is assumed for all nodes.

ProfileThresholdModel.**__init__**(*graph*)

Model Constructor

Parameters **graph** – A networkx graph object

ProfileThresholdModel.**set_initial_status**(*self*, *configuration*)

Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

ProfileThresholdModel.**reset**(*self*)

Reset the simulation setting the actual status to the initial configuration.

Describe

ProfileThresholdModel.**get_info**(*self*)

Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

ProfileThresholdModel.**get_status_map**(*self*)

Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

ProfileThresholdModel.**iteration**(*self*)

Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

ProfileThresholdModel.**iteration_bunch**(*self*, *bunch_size*)

Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Profile Threshold model simulation on a random graph: we set the initial infected node set to the 10% of the overall population, assign a profile of 0.25 and a threshold of 0.15 to all the nodes.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.ProfileThresholdModel(g)
config = mc.Configuration()
config.add_model_parameter('blocked', 0)
config.add_model_parameter('adopter_rate', 0)
config.add_model_parameter('fraction_infected', 0.1)

# Setting nodes parameters
threshold = 0.15
profile = 0.25
for i in g.nodes():
    config.add_node_configuration("threshold", i, threshold)
    config.add_node_configuration("profile", i, profile)

model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Opinion Dynamics

A different field related with modelling social behaviour is that of opinion dynamics.

Recent years have witnessed the introduction of a wide range of models that attempt to explain how opinions form in a population[5], taking into account various social theories (e.g. bounded confidence[6] or social impact[7]).

These models have a lot in common with those seen in epidemics and spreading. In general, individuals are modelled as agents with a state and connected by a social network.

[5] A. Sırbu, V. Loreto, V. D. Servedio, and F. Tria, "Opinion dynamics: Models, extensions and external effects," in Participatory Sensing, Opinions and Collective Awareness. Springer International Publishing, 2017, pp. 363–401.

[6] G. Deffuant, D. Neau, F. Amblard, and G. Weisbuch, "Mixing beliefs among interacting agents," Advances in Complex Systems, vol. 3, no. 4, pp. 87–98, 2000.

[7] K. Sznajd-Weron and J. Sznajd, "Opinion evolution in closed community," International Journal of Modern Physics C, vol. 11, pp. 1157–1165, 2001.

The social links can be represented by a complete graph (mean field models) or by more realistic complex networks, similar to epidemics and spreading.

The state is typically represented by variables, that can be discrete (similar to the case of spreading), but also continuous, representing for instance a probability to choose one option or another[8] . The state of individuals changes in time, based on a set of update rules, mainly through interaction with the neighbours.

While in many spreading and epidemics models this change is irreversible (susceptible to infected), in opinion dynamics the state can oscillate freely between the possible values, simulating thus how opinions change in reality.

A different important aspect in opinion dynamics is external information, which can be interpreted as the effect of mass media. In general external information is represented as a static individual with whom all others can interact, again present also in spreading models. Hence, it is clear that the two model categories have enough in common to be implemented under a common framework, which is why we introduced both in our framework.

In `NDlib` are implemented the following **Opinion Dynamics** models:

Voter

The Voter model is one of the simplest models of opinion dynamics, originally introduced to analyse competition of species[1] and soon after applied to model elections[2].

The model assumes the opinion of an individual to be a discrete variable ± 1.

The state of the population varies based on a very simple update rule: at each iteration, a random individual is selected, who then copies the opinion of one random neighbour.

Starting from any initial configuration, on a complete network the entire population converges to consensus on one of the two options. The probability that consensus is reached on opinion +1 is equal to the initial fraction of individuals holding that opinion[3].

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

Parameters

The initial infection status can be defined via:

[8] A. Sırbu, V. Loreto, V. D. Servedio, and F. Tria, "Opinion dynamics with disagreement and modulated information," Journal of Statistical Physics, pp. 1–20, 2013.

[1] P. Clifford and A. Sudbury, "A model for spatial conflict," Biometrika, vol. 60, no. 3, pp. 581–588, 1973.

[2] R. Holley and T. Liggett, "Ergodic theorems for weakly interacting infinite systems and the voter model," Ann. Probab., vol. 3, no. 4, pp. 643–663, Aug 1975.

[3] P.L.Krapivsky,S.Redner,andE.Ben-Naim,Akineticviewofstatistical physics. Cambridge University Press, 2010.

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The initial blocked nodes can be defined via:

- **percentage_blocked**: Model Parameter, float in [0, 1]
- **Blocked**: Status Parameter, set of nodes

In both cases, the two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.opinions.VoterModel.**VoterModel**(*graph*, *seed=None*)

VoterModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

VoterModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

VoterModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

VoterModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

VoterModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

VoterModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

VoterModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Voter model simulation on a random graph: we set the initial infected node set to the 10% of the overall population.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.opinions as op

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = op.VoterModel(g)
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Q-Voter

The Q-Voter model was introduced as a generalisation of discrete opinion dynamics models[1].

Here, N individuals hold an opinion ± 1. At each time step, a set of **q** neighbours are chosen and, if they agree, they influence one neighbour chosen at random, i.e. this agent copies the opinion of the group. If the group does not agree, the agent flips its opinion with probability ϵ.

It is clear that the voter and Sznajd models are special cases of this more recent model (q = 1,ϵ = 0 and q = 2,ϵ = 0).

Analytic results for q 3 validate the numerical results obtained for the special case models, with transitions from a ordered phase (small ϵ) to a disordered one (large ϵ). For q > 3, a new type of transition between the two phases appears, which consist of passing through an intermediate regime where the final state depends on the initial condition. We implemented in NDlib the model with ϵ = 0.

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

[1] C. Castellano, M. A. Munoz, and R. Pastor-Satorras, "The non-linear q-voter model," Physical Review E, vol. 80, p. 041129, 2009.

Parameters

Name	Type	Value Type	Default	Mandatory	Description
q	Model	int in [0, V(G)]		True	Number of neighbours

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.opinions.QVoterModel.**QVoterModel**(*graph*, *seed=None*)
Node Parameters to be specified via ModelConfig

Parameters **q** – the number of neighbors that affect the opinion of a node

QVoterModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

QVoterModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

QVoterModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

QVoterModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

QVoterModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

QVoterModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

QVoterModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Q-Voter model simulation on a random graph: we set the initial infected node set to the 10% of the overall population and the number **q** of influencing neighbors equals to 5.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.opinions as op

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = op.QVoterModel(g)
config = mc.Configuration()
config.add_model_parameter("q", 5)
config.add_model_parameter('fraction_infected', 0.1)

model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Majority Rule

The Majority Rule model is a discrete model of opinion dynamics, proposed to describe public debates[1].

Agents take discrete opinions ± 1, just like the Voter model. At each time step a group of **r** agents is selected randomly and they all take the majority opinion within the group.

The group size can be fixed or taken at each time step from a specific distribution. If **r** is odd, then the majority opinion is always defined, however if **r** is even there could be tied situations. To select a prevailing opinion in this case, a bias in favour of one opinion (+1) is introduced.

This idea is inspired by the concept of social inertia[2].

[1] S.Galam, "Minority opinion spreading in random geometry." Eur.Phys. J. B, vol. 25, no. 4, pp. 403–406, 2002.
[2] R.Friedman and M.Friedman, "The Tyranny of the Status Quo." Orlando, FL, USA: Harcourt Brace Company, 1984.

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
q	Model	int in [0, V(G)]		True	Number of neighbours

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.opinions.MajorityRuleModel.**MajorityRuleModel**(*graph*, *seed=None*)

MajorityRuleModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

MajorityRuleModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

MajorityRuleModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

MajorityRuleModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

MajorityRuleModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

MajorityRuleModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

MajorityRuleModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Majority Rule model simulation on a random graph: we set the initial infected node set to the 10% of the overall population.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.opinions as op

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = op.MajorityRuleModel(g)
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Sznajd

The Sznajd model[1] is a variant of spin model employing the theory of social impact, which takes into account the fact that a group of individuals with the same opinion can influence their neighbours more than one single individual.

[1] K. Sznajd-Weron and J. Sznajd, "Opinion evolution in closed community," International Journal of Modern Physics C, vol. 11, pp. 1157–1165, 2001.

In the original model the social network is a 2-dimensional lattice, however we also implemented the variant on any complex networks.

Each agent has an opinion $\sigma i = \pm 1$. At each time step, a pair of neighbouring agents is selected and, if their opinion coincides, all their neighbours take that opinion.

The model has been shown to converge to one of the two agreeing stationary states, depending on the initial density of up-spins (transition at 50% density).

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

Parameters

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.opinions.SznajdModel.**SznajdModel**(*graph*, *seed=None*)

SznajdModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

SznajdModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

SznajdModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

SznajdModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

SznajdModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

SznajdModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

SznajdModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Sznajd model simulation on a random graph: we set the initial infected node set to the 10% of the overall population.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.opinions as op

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = op.SznajdModel(g)
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Cognitive Opinion Dynamics

The Cognitive Opinion Dynamics model was introduced in[1], which models the state of individuals taking into account several cognitively-grounded variables.

The aim of the model is to simulate response to risk in catastrophic events in the presence of external (institutional) information.

The individual opinion is modelled as a continuous variable **Oi** [0, 1], representing the degree of perception of the risk (how probable it is that the catastrophic event will actually happen).

This opinion evolves through interactions with neighbours and external information, based on four internal variables for each individual i:

- risk sensitivity (Ri {1, 0, 1}),
- tendency to inform others (βi [0,1]),
- trust in institutions (Ti [0,1]), and
- trust in peers (Πi = 1 Ti).

These values are generated when the population is initialised and stay fixed during the simulation.

The update rules define how **Oi** values change in time.

The model was shown to be able to reproduce well various real situations. In particular, it is visible that risk sensitivity is more important than trust in institutional information when it comes to evaluating risky situations.

Statuses

Node statuses are continuous values in [0,1].

[1] D. Vilone, F. Giardini, M. Paolucci, and R. Conte, "Reducing individuals' risk sensitiveness can promote positive and non-alarmist views about catastrophic events in an agent-based simulation," arXiv preprint arXiv:1609.04566, 2016.

Parameters

Name	Type	Value Type	Default	Mandatory	Description
I	Model	float in [0, 1]		True	External information
T_range_min	Model	float in [0, 1]		True	Minimum of the range of initial values for T
T_range_max	Model	float in [0, 1]		True	Maximum of the range of initial values for T
B_range_min	Model	float in [0, 1]		True	Minimum of the range of initial values for B
B_range_max	Model	float in [0, 1]		True	Maximum of the range of initial values for B
R_fraction_negative	Model	float in [0, 1]		True	Fraction of nodes having R=-1
R_fraction_neutral	Model	float in [0, 1]		True	Fraction of nodes having R=0
R_fraction_positive	Model	float in [0, 1]		True	Fraction of nodes having R=1

The following relation should hold: `R_fraction_negative+R_fraction_neutral+R_fraction_positive=1`. To achieve this, the fractions selected will be normalised to sum 1.

The initial state is generated randomly uniformly from the domain defined by model parameters.

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.opinions.CognitiveOpDynModel.**CognitiveOpDynModel**(*graph*, *seed=None*)

Model Parameters to be specified via ModelConfig

Parameters

- **I** – external information value in [0,1]
- **T_range_min** – the minimum of the range of initial values for T. Range [0,1].
- **T_range_max** – the maximum of the range of initial values for T. Range [0,1].
- **B_range_min** – the minimum of the range of initial values for B. Range [0,1]
- **B_range_max** – the maximum of the range of initial values for B. Range [0,1].
- **R_fraction_negative** – fraction of individuals having the node parameter R=-1.

- **R_fraction_positive** – fraction of individuals having the node parameter R=1
- **R_fraction_neutral** – fraction of individuals having the node parameter R=0

The following relation should hold: R_fraction_negative+R_fraction_neutral+R_fraction_positive=1. To achieve this, the fractions selected will be normalised to sum 1. Node states are continuous values in [0,1].

The initial state is generated randomly uniformly from the domain defined by model parameters.

CognitiveOpDynModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

CognitiveOpDynModel.**set_initial_status**(*self*, *configuration*)
Override behaviour of methods in class DiffusionModel. Overwrites initial status using random real values. Generates random node profiles.

CognitiveOpDynModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

CognitiveOpDynModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

CognitiveOpDynModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

CognitiveOpDynModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

CognitiveOpDynModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a Cognitive Opinion Dynamics model simulation on a random graph: we set the initial infected node set to the 10% of the overall population, the external information value to 015, the B and T intervals equal to [0,1] and the fraction of positive/neutral/infected equal to 1/3.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.opinions as op

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = op.CognitiveOpDynModel(g)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter("I", 0.15)
config.add_model_parameter("B_range_min", 0)
config.add_model_parameter("B_range_max", 1)
config.add_model_parameter("T_range_min", 0)
config.add_model_parameter("T_range_max", 1)
config.add_model_parameter("R_fraction_negative", 1.0 / 3)
config.add_model_parameter("R_fraction_neutral", 1.0 / 3)
config.add_model_parameter("R_fraction_positive", 1.0 / 3)
config.add_model_parameter('fraction_infected', 0.1)
model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Algorithmic Bias

Note: The Algorithmic Bias model will be officially released in NDlib 4.0.1

The Algorithmic Bias model considers a population of individuals, where each individual holds a continuous opinion in the interval [0,1]. Individuals are connected by a social network, and interact pairwise at discrete time steps. The interacting pair is selected from the population at each time point in such a way that individuals that have close opinion values are selected more often, to simulate algorithmic bias. The parameter gamma controls how large this effect is. Specifically, the first individual in the interacting pair is selected randomly, while the second individual is selected based on a probability that decreases with the distance from the opinion of the first individual, i.e. directly proportional with the distance raised to the power -gamma.

After interaction, the two opinions may change, depending on a so called bounded confidence parameter, epsilon. This can be seen as a measure of the open-mindedness of individuals in a population. It defines a threshold on the distance between the opinion of the two individuals, beyond which communication between individuals is not possible due to conflicting views. Thus, if the distance between the opinions of the selected individuals is lower than epsilon, the two individuals adopt their average opinion. Otherwise nothing happens.

Note: setting gamma=0 reproduce the results for the Deffuant model.

Statuses

Node statuses are continuous values in [0,1].

Parameters

Name	Type	Value Type	Default	Mandatory	Description
epsilon	Model	float in [0, 1]		True	Bounded confidence threshold
gamma	Model	int in [0, 100]		True	Algorithmic bias

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.opinions.AlgorithmicBiasModel.**AlgorithmicBiasModel**(*graph*, *seed=None*)

Model Parameters to be specified via ModelConfig

Parameters

- **epsilon** – bounded confidence threshold from the Deffuant model, in [0,1]
- **gamma** – strength of the algorithmic bias, positive, real

Node states are continuous values in [0,1].

The initial state is generated randomly uniformly from the domain [0,1].

AlgorithmicBiasModel.**__init__**(*graph*)

Model Constructor

Parameters **graph** – A networkx graph object

AlgorithmicBiasModel.**set_initial_status**(*self*, *configuration*)

Override behaviour of methods in class DiffusionModel. Overwrites initial status using random real values.

AlgorithmicBiasModel.**reset**(*self*)

Reset the simulation setting the actual status to the initial configuration.

Describe

AlgorithmicBiasModel.**get_info**(*self*)

Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

AlgorithmicBiasModel.**get_status_map**(*self*)

Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

AlgorithmicBiasModel.**iteration**(*self*)

Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

AlgorithmicBiasModel.**iteration_bunch**(*self*, *bunch_size*)
Execute a bunch of model iterations

Parameters

- **bunch_size** – the number of iterations to execute
- **node_status** – if the incremental node status has to be returned.

Returns a list containing for each iteration a dictionary {"iteration": iteration_id, "status": dictionary_node_to_status}

Example

In the code below is shown an example of instantiation and execution of a AlgorithmicBiasModel model simulation on a random graph: we set the initial infected node set to the 10% of the overall population.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.opinions as op

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = op.AlgorithmicBiasModel(g)

# Model configuration
config = mc.Configuration()
config.add_model_parameter("epsilon", 0.32)
config.add_model_parameter("gamma", 1)
model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(200)
```

Dynamic Network Models

Network topology may evolve as time goes by.

In order to automatically leverage network dynamics NDlib enables the definition of diffusion models that work on *Snapshot Graphs* as well as on *Interaction Networks*.

In particular NDlib implements dynamic network versions of the following models:

SI

The SI model was introduced in 1927 by Kermack[1].

[1] W. O. Kermack and A. McKendrick, "A Contribution to the Mathematical Theory of Epidemics," Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, vol. 115, no. 772, pp. 700–721, Aug. 1927.

In this model, during the course of an epidemics, a node is allowed to change its status only from **Susceptible** (S) to **Infected** (I).

The model is instantiated on a graph having a non-empty set of infected nodes.

SI assumes that if, during a generic iteration, a susceptible node comes into contact with an infected one, it becomes infected with probability β: once a node becomes infected, it stays infected (the only transition allowed is S→I).

The dSI implementation assumes that the process occurs on a directed/undirected dynamic network; this model was introduced by Milli et al. in 2018[2].

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
beta	Model	float in [0, 1]		True	Infection probability

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.dynamic.DynSIModel.**DynSIModel**(*graph*, *seed=None*)
Model Parameters to be specified via ModelConfig

Parameters **beta** – The infection rate (float value in [0,1])

DynSIModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A dynetx graph object

DynSIModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

[2] Letizia Milli, Giulio Rossetti, Fosca Giannotti, Dino Pedreschi. "Diffusive Phenomena in Dynamic Networks: a data-driven study". Accepted to International Conference on Complex Networks (CompleNet), 2018, Boston.

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

DynSIModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

DynSIModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

DynSIModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

DynSIModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

DynSIModel.**execute_snapshots**(*bunch_size*, *node_status*)

DynSIModel.**execute_iterations**(*node_status*)

Example

In the code below is shown an example of instantiation and execution of an DynSI simulation on a dynamic random graph: we set the initial set of infected nodes as 5% of the overall population and a probability of infection of 1%.

```
import networkx as nx
import dynetx as dn
import ndlib.models.ModelConfig as mc
import ndlib.models.dynamic as dm
from past.builtins import xrange

# Dynamic Network topology
dg = dn.DynGraph()

for t in xrange(0, 3):
    g = nx.erdos_renyi_graph(200, 0.05)
    dg.add_interactions_from(g.edges(), t)

# Model selection
model = dm.DynSIModel(dg)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('beta', 0.01)
config.add_model_parameter("fraction_infected", 0.1)
model.set_initial_status(config)
```

(continues on next page)

(continued from previous page)

```
# Simulate snapshot based execution
iterations = model.execute_snapshots()

# Simulation interaction graph based execution
iterations = model.execute_iterations()
```

SIS

The SIS model was introduced in 1927 by Kermack[1].

In this model, during the course of an epidemics, a node is allowed to change its status from **Susceptible** (S) to **Infected** (I).

The model is instantiated on a graph having a non-empty set of infected nodes.

SIS assumes that if, during a generic iteration, a susceptible node comes into contact with an infected one, it becomes infected with probability beta, than it can be switch again to susceptible with probability lambda (the only transition allowed are S→I→S).

The dSIS implementation assumes that the process occurs on a directed/undirected dynamic network.

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
beta	Model	float in [0, 1]		True	Infection probability
lambda	Model	float in [0, 1]		True	Recovery probability

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

[1] W. O. Kermack and A. McKendrick, "A Contribution to the Mathematical Theory of Epidemics," Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, vol. 115, no. 772, pp. 700–721, Aug. 1927

Configure

class ndlib.models.dynamic.DynSISModel.**DynSISModel**(*graph*, *seed=None*)
Model Parameters to be specified via ModelConfig

Parameters

- **beta** – The infection rate (float value in [0,1])
- **lambda** – The recovery rate (float value in [0,1])

DynSISModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

DynSISModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

DynSISModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

DynSISModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

DynSISModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

DynSISModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

DynSISModel.**execute_snapshots**(*bunch_size*, *node_status*)

DynSISModel.**execute_iterations**(*node_status*)

Example

In the code below is shown an example of instantiation and execution of an DynSIS simulation on a dynamic random graph: we set the initial set of infected nodes as 5% of the overall population, a probability of infection of 1%, and a probability of recovery of 1%.

```
import networkx as nx
import dynetx as dn
import ndlib.models.ModelConfig as mc
import ndlib.models.dynamic as dm
from past.builtins import xrange

# Dynamic Network topology
dg = dn.DynGraph()

for t in xrange(0, 3):
    g = nx.erdos_renyi_graph(200, 0.05)
    dg.add_interactions_from(g.edges(), t)

# Model selection
model = dm.DynSISModel(dg)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('beta', 0.01)
config.add_model_parameter('lambda', 0.01)
config.add_model_parameter("fraction_infected", 0.1)
model.set_initial_status(config)

# Simulate snapshot based execution
iterations = model.execute_snapshots()

# Simulation interaction graph based execution
iterations = model.execute_iterations()
```

SIR

The SIR model was introduced in 1927 by Kermack[1].

In this model, during the course of an epidemics, a node is allowed to change its status from **Susceptible** (S) to **Infected** (I), then to **Removed** (R).

The model is instantiated on a graph having a non-empty set of infected nodes.

SIR assumes that if, during a generic iteration, a susceptible node comes into contact with an infected one, it becomes infected with probability beta, than it can be switch to removed with probability gamma (the only transition allowed are S→I→R).

The dSIR implementation assumes that the process occurs on a directed/undirected dynamic network; this model was introduced by Milli et al. in 2018[2].

Statuses

During the simulation a node can experience the following statuses:

[1] W. O. Kermack and A. McKendrick, "A Contribution to the Mathematical Theory of Epidemics," Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character, vol. 115, no. 772, pp. 700–721, Aug. 1927

[2] Letizia Milli, Giulio Rossetti, Fosca Giannotti, Dino Pedreschi. "Diffusive Phenomena in Dynamic Networks: a data-driven study". Accepted to International Conference on Complex Networks (CompleNet), 2018, Boston.

Name	Code
Susceptible	0
Infected	1
Removed	2

Parameters

Name	Type	Value Type	Default	Mandatory	Description
beta	Model	float in [0, 1]		True	Infection probability
gamma	Model	float in [0, 1]		True	Removal probability

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.dynamic.DynSIRModel.**DynSIRModel**(*graph*, *seed=None*)
Model Parameters to be specified via ModelConfig

Parameters

- **beta** – The infection rate (float value in [0,1])
- **gamma** – The recovery rate (float value in [0,1])

DynSIRModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

DynSIRModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

DynSIRModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

DynSIRModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

`DynSIRModel.`**`get_status_map`**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

`DynSIRModel.`**`iteration`**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

`DynSIRModel.`**`execute_snapshots`**(*bunch_size*, *node_status*)

`DynSIRModel.`**`execute_iterations`**(*node_status*)

Example

In the code below is shown an example of instantiation and execution of an DynSIR simulation on a dynamic random graph: we set the initial set of infected nodes as 5% of the overall population, a probability of infection of 1%, and a removal probability of 1%.

```
import networkx as nx
import dynetx as dn
import ndlib.models.ModelConfig as mc
import ndlib.models.dynamic as dm
from past.builtins import xrange

# Dynamic Network topology
dg = dn.DynGraph()

for t in xrange(0, 3):
    g = nx.erdos_renyi_graph(200, 0.05)
    dg.add_interactions_from(g.edges(), t)

# Model selection
model = dm.DynSIRModel(dg)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('beta', 0.01)
config.add_model_parameter('gamma', 0.01)
config.add_model_parameter("fraction_infected", 0.1)
model.set_initial_status(config)

# Simulate snapshot based execution
iterations = model.execute_snapshots()

# Simulation interaction graph based execution
iterations = model.execute_iterations()
```

Kertesz Threshold

The Kertesz Threshold model was introduced in 2015 by Ruan et al.[1] and it is an extension of the Watts threshold model[2].

The authors extend the classical model introducing a density **r** of blocked nodes – nodes which are immune to social influence – and a probability of spontaneous adoption **p** to capture external influence.

Thus, the model distinguishes three kinds of node: **Blocked** (B), **Susceptible** (S) and **Adoptiong** (A). The latter class breaks into two categories: vulnerable and stable nodes. A node can adopt either under its neighbors' influence, or spontaneously, due to endogenous effects.

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1
Blocked	-1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
adopter_rate	Model	float in [0, 1]	0	False	Exogenous adoption rate
percentage_blocked	Model	float in [0, 1]	0.1	False	Blocked nodes
threshold	Node	float in [0, 1]	0.1	False	Individual threshold

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The initial blocked nodes can be defined via:

- **percentage_blocked**: Model Parameter, float in [0, 1]
- **Blocked**: Status Parameter, set of nodes

In both cases, the two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

[1] 26. Ruan, G. In iguez, M. Karsai, and J. Kertesz, "Kinetics of social contagion," Phys. Rev. Lett., vol. 115, p. 218702, Nov 2015.

[2] D. J. Watts, "A simple model of global cascades on random networks," Proceedings of the National Academy of Sciences, vol. 99, no. 9, pp. 5766–5771, 2002.

Configure

class ndlib.models.dynamic.DynKerteszThresholdModel.**DynKerteszThresholdModel**(*graph*, *seed=None*)

Node Parameters to be specified via ModelConfig

Parameters **profile** – The node profile. As default a value of 0.1 is assumed for all nodes.

DynKerteszThresholdModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

DynKerteszThresholdModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

DynKerteszThresholdModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

DynKerteszThresholdModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

DynKerteszThresholdModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

DynKerteszThresholdModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

DynKerteszThresholdModel.**execute_snapshots**(*bunch_size*, *node_status*)

DynKerteszThresholdModel.**execute_iterations**(*node_status*)

Example

In the code below is shown an example of instantiation and execution of a Kertesz Threshold model simulation on a random graph: we set the initial infected as well blocked node sets equals to the 10% of the overall population, assign a threshold of 0.25 to all the nodes and impose an probability of spontaneous adoptions of 40%.

```
import networkx as nx
import dynetx as dn
import ndlib.models.ModelConfig as mc
import ndlib.models.dynamic as dm

# Dynamic Network topology
dg = dn.DynGraph()

for t in past.builtins.xrange(0, 3):
    g = nx.erdos_renyi_graph(200, 0.05)
    dg.add_interactions_from(g.edges(), t)

# Model selection
model = dm.DynKerteszThresholdModel(g)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('adopter_rate', 0.4)
config.add_model_parameter('percentage_blocked', 0.1)
config.add_model_parameter('fraction_infected', 0.1)

# Setting node parameters
threshold = 0.25
for i in g.nodes():
    config.add_node_configuration("threshold", i, threshold)

model.set_initial_status(config)

# Simulate snapshot based execution
iterations = model.execute_snapshots()
```

Profile

The Profile model, introduced by Milli et al. in[1], assumes that the diffusion process is only apparent; each node decides to adopt or not a given behavior – once known its existence – only on the basis of its own interests.

In this scenario the peer pressure is completely ruled out from the overall model: it is not important how many of its neighbors have adopted a specific behaviour, if the node does not like it, it will not change its interests.

Each node has its own profile describing how many it is likely to accept a behaviour similar to the one that is currently spreading.

The diffusion process starts from a set of nodes that have already adopted a given behaviour S:

- for each of the susceptible nodes' in the neighborhood of a node u in S an unbalanced coin is flipped, the unbalance given by the personal profile of the susceptible node;
- if a positive result is obtained the susceptible node will adopt the behaviour, thus becoming infected.
- if the **blocked** status is enabled, after having rejected the adoption with probability `blocked` a node becomes immune to the infection.
- every iteration `adopter_rate` percentage of nodes spontaneous became infected to endogenous effects.

[1] Milli, L., Rossetti, G., Pedreschi, D., & Giannotti, F. (2018). Active and passive diffusion processes in complex networks. Applied network science, 3(1), 42.

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1
Blocked	-1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
profile	Node	float in [0, 1]	0.1	False	Node profile
blocked	Model	float in [0, 1]	0	False	Blocked nodes
adopter_rate	Model	float in [0, 1]	0	False	Autonomous adoption

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.dynamic.DynProfileModel.**DynProfileModel**(*graph*, *seed=None*)

Node Parameters to be specified via ModelConfig

Parameters **profile** – The node profile. As default a value of 0.1 is assumed for all nodes.

DynProfileModel.**__init__**(*graph*)
Model Constructor

Parameters **graph** – A networkx graph object

DynProfileModel.**set_initial_status**(*self*, *configuration*)
Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

DynProfileModel.**reset**(*self*)
Reset the simulation setting the actual status to the initial configuration.

Describe

DynProfileModel.**get_info**(*self*)
Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

DynProfileModel.**get_status_map**(*self*)
Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

DynProfileModel.**iteration**(*self*)
Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

DynProfileModel.**execute_snapshots**(*bunch_size*, *node_status*)

NB: the ''execute_iterations()'' method is unavailable for this model (along with other thresholded models).

Example

In the code below is shown an example of instantiation and execution of a Profile model simulation on a random graph: we set the initial infected node set to the 10% of the overall population and assign a profile of 0.25 to all the nodes.

```
import networkx as nx
import dynetx as dn
import ndlib.models.ModelConfig as mc
import ndlib.models.dynamic as dm
from past.builtins import xrange

# Dynamic Network topology
dg = dn.DynGraph()

for t in xrange(0, 3):
    g = nx.erdos_renyi_graph(200, 0.05)
    dg.add_interactions_from(g.edges(), t)

# Model selection
model = dm.DynProfileModel(dg)
config = mc.Configuration()
config.add_model_parameter('blocked', 0)
config.add_model_parameter('adopter_rate', 0)
config.add_model_parameter('fraction_infected', 0.1)

# Setting nodes parameters
profile = 0.15
for i in g.nodes():
    config.add_node_configuration("profile", i, profile)

model.set_initial_status(config)
```

(continues on next page)

(continued from previous page)

```
# Simulate snapshot based execution
iterations = model.execute_snapshots()
```

Threshold

The Profile-Threshold model, introduced by Milli et al. in[1], assumes the existence of node profiles that act as preferential schemas for individual tastes but relax the constraints imposed by the Profile model by letting nodes influenceable via peer pressure mechanisms.

The peer pressure is modeled with a threshold.

The diffusion process starts from a set of nodes that have already adopted a given behaviour S:

- for each of the susceptible node an unbalanced coin is flipped if the percentage of its neighbors that are already infected excedes its threhosld. As in the Profile Model the coin unbalance is given by the personal profile of the susceptible node;
- if a positive result is obtained the susceptible node will adopt the behaviour, thus becoming infected.
- if the **blocked** status is enabled, after having rejected the adoption with probability `blocked` a node becomes immune to the infection.
- every iteration `adopter_rate` percentage of nodes spontaneous became infected to endogenous effects.

Statuses

During the simulation a node can experience the following statuses:

Name	Code
Susceptible	0
Infected	1
Blocked	-1

Parameters

Name	Type	Value Type	Default	Mandatory	Description
threshold	Node	float in [0, 1]	0.1	False	Individual threshold
profile	Node	float in [0, 1]	0.1	False	Node profile
blocked	Model	float in [0, 1]	0	False	Blocked nodes
adopter_rate	Model	float in [0, 1]	0	False	Autonomous adoption

The initial infection status can be defined via:

- **fraction_infected**: Model Parameter, float in [0, 1]
- **Infected**: Status Parameter, set of nodes

The two options are mutually exclusive and the latter takes precedence over the former.

[1] Milli, L., Rossetti, G., Pedreschi, D., & Giannotti, F. (2018). Active and passive diffusion processes in complex networks. Applied network science, 3(1), 42.

Methods

The following class methods are made available to configure, describe and execute the simulation:

Configure

class ndlib.models.dynamic.DynProfileThresholdModel.**DynProfileThresholdModel**(*graph*, *seed=None*)

Node Parameters to be specified via ModelConfig

Parameters

- **profile** – The node profile. As default a value of 0.1 is assumed for all nodes.
- **threshold** – The node threshold. As default a value of 0.1 is assumed for all nodes.

DynProfileThresholdModel.**__init__**(*graph*)

Model Constructor

Parameters **graph** – A networkx graph object

DynProfileThresholdModel.**set_initial_status**(*self*, *configuration*)

Set the initial model configuration

Parameters **configuration** – a `ndlib.models.ModelConfig.Configuration` object

DynProfileThresholdModel.**reset**(*self*)

Reset the simulation setting the actual status to the initial configuration.

Describe

DynProfileThresholdModel.**get_info**(*self*)

Describes the current model parameters (nodes, edges, status)

Returns a dictionary containing for each parameter class the values specified during model configuration

DynProfileThresholdModel.**get_status_map**(*self*)

Specify the statuses allowed by the model and their numeric code

Returns a dictionary (status->code)

Execute Simulation

DynProfileThresholdModel.**iteration**(*self*)

Execute a single model iteration

Returns Iteration_id, Incremental node status (dictionary node->status)

DynProfileThresholdModel.**execute_snapshots**(*bunch_size*, *node_status*)

NB: the ‘‘execute_iterations()‘‘ method is unavailable for this model (along with other thresholded models).

Example

In the code below is shown an example of instantiation and execution of a Profile Threshold model simulation on a random graph: we set the initial infected node set to the 10% of the overall population, assign a profile of 0.25 and a threshold of 0.15 to all the nodes.

```
import networkx as nx
import dynetx as dn
import ndlib.models.ModelConfig as mc
import ndlib.models.dynamic as dm
from past.builtins import xrange

# Dynamic Network topology
dg = dn.DynGraph()

for t in xrange(0, 3):
    g = nx.erdos_renyi_graph(200, 0.05)
    dg.add_interactions_from(g.edges(), t)

# Model selection
model = dm.DynProfileThresholdModel(dg)
config = mc.Configuration()
config.add_model_parameter('blocked', 0)
config.add_model_parameter('adopter_rate', 0)
config.add_model_parameter('fraction_infected', 0.1)

# Setting nodes parameters
threshold = 0.15
profile = 0.25
for i in g.nodes():
    config.add_node_configuration("threshold", i, threshold)
    config.add_node_configuration("profile", i, profile)

model.set_initial_status(config)


# Simulate snapshot based execution
iterations = model.execute_snapshots()
```

1.5.2 Model Configuration

NDlib adopts a peculiar approach to specify the configuration of expetiments. It employs a centralyzed system that take care of:

1. Describe a **common syntax** for model configuration;
2. Provide an interface to set the **initial conditions** of an experiment (nodes/edges properties, initial nodes statuses)

ModelConfig

The ModelConfig object is the common interface used to set up simulation experiments.

class ndlib.models.ModelConfig.**Configuration**
Configuration Object

It allows to specify four categories of experiment configurations:

1. Model configuration
2. Node Configuration
3. Edge Configuration
4. Initial Status

Every diffusion model has its own parameters (as defined in its reference page).

Model Configuration

Model configuration involves the instantiation of both the *mandatory* and *optional* parameters of the chosen diffusion model.

`Configuration.`**`add_model_parameter`**(*self*, *param_name*, *param_value*)
Set a Model Parameter

Parameters

- **`param_name`** – parameter identifier (as specified by the chosen model)
- **`param_value`** – parameter value

Model parameters can be setted as in the following example:

```
import ndlib.models.ModelConfig as mc

# Model Configuration
config = mc.Configuration()
config.add_model_parameter("beta", 0.15)
```

The only model parameter common to all the diffusive approaches is `fraction_infected` that allows to specify the ratio of infected nodes at the beginning of the simulation.

Node Configuration

Node configuration involves the instantiation of both the *mandatory* and *optional* parameters attached to individual nodes.

`Configuration.`**`add_node_configuration`**(*self*, *param_name*, *node_id*, *param_value*)
Set a parameter for a given node

Parameters

- **`param_name`** – parameter identifier (as specified by the chosen model)
- **`node_id`** – node identifier
- **`param_value`** – parameter value

`Configuration.`**`add_node_set_configuration`**(*self*, *param_name*, *node_to_value*)
Set Nodes parameter

Parameters

- **`param_name`** – parameter identifier (as specified by the chosen model)
- **`node_to_value`** – dictionary mapping each node a parameter value

Node parameters can be set as in the following example:

```
import ndlib.models.ModelConfig as mc

# Model Configuration
config = mc.Configuration()

threshold = 0.25
for i in g.nodes():
    config.add_node_configuration("threshold", i, threshold)
```

Edge Configuration

Edge configuration involves the instantiation of both the *mandatory* and *optional* parameters attached to individual edges.

`Configuration.`**`add_edge_configuration`**(*self*, *param_name*, *edge*, *param_value*)
Set a parameter for a given edge

Parameters

- **`param_name`** – parameter identifier (as specified by the chosen model)
- **`edge`** – edge identifier
- **`param_value`** – parameter value

`Configuration.`**`add_edge_set_configuration`**(*self*, *param_name*, *edge_to_value*)
Set Edges parameter

Parameters

- **`param_name`** – parameter identifier (as specified by the chosen model)
- **`edge_to_value`** – dictionary mapping each edge a parameter value

Edge parameters can be set as in the following example:

```
import ndlib.models.ModelConfig as mc

# Model Configuration
config = mc.Configuration()

threshold = 0.25
for i in g.nodes():
    config.add_edge_configuration("threshold", i, threshold)
```

Status Configuration

Status configuration allows to specify explicitly the status of a set of nodes at the beginning of the simulation.

`Configuration.`**`add_model_initial_configuration`**(*self*, *status_name*, *nodes*)
Set initial status for a set of nodes

Parameters

- **`status_name`** – status to be set (as specified by the chosen model)
- **`nodes`** – list of affected nodes

Node statuses can be set as in the following example:

```
import ndlib.models.ModelConfig as mc

# Model Configuration
config = mc.Configuration()

infected_nodes = [0, 1, 2, 3, 4, 5]
config.add_model_initial_configuration("Infected", infected_nodes)
```

Explicit status specification takes priority over the percentage specification expressed via model definition (e.g. `fraction_infected`).

Only the statuses implemented by the chosen model can be used to specify initial configurations of nodes.

1.5.3 NDlib Utils

The `ndlib.utils` module contains facilities that extend the simulation framework (i.e., automated multiple executions).

Model Multiple Executions

`dlib.utils.multi_runs` allows the parallel execution of multiple instances of a given model starting from different initial infection conditions.

The initial infected nodes for each instance of the model can be specified either:

- by the "fraction_infected" model parameter, or
- explicitly through a list of `n` sets of nodes (where `n` is the number of executions required).

In the first scenario "fraction_infected" nodes will be sampled independently for each model execution.

Results of `dlib.utils.multi_runs` can be feed directly to all the visualization facilities exposed by `ndlib.viz`.

`ndlib.utils.`**`multi_runs`**(*model*, *execution_number*, *iteration_number*, *infection_sets*, *nprocesses*)

Multiple executions of a given model varying the initial set of infected nodes

Parameters

- **`model`** – a configured diffusion model
- **`execution_number`** – number of instantiations
- **`iteration_number`** – number of iterations per execution
- **`infection_sets`** – predefined set of infected nodes sets
- **`nprocesses`** – number of processes. Default values cpu number.

Returns resulting trends for all the executions

Example

Randomly selection of initial infection sets

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
from ndlib.utils import multi_runs

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model1 = ep.SIRModel(g)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('beta', 0.001)
config.add_model_parameter('gamma', 0.01)
config.add_model_parameter("fraction_infected", 0.05)
model1.set_initial_status(config)

# Simulation multiple execution
trends = multi_runs(model1, execution_number=10, iteration_number=100, infection_
↪sets=None, nprocesses=4)
```

Specify initial infection sets

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
from ndlib.utils import multi_runs

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model1 = ep.SIRModel(g)

# Model Configuration
config = mc.Configuration()
config.add_model_parameter('beta', 0.001)
config.add_model_parameter('gamma', 0.01)
model1.set_initial_status(config)

# Simulation multiple execution
infection_sets = [(1, 2, 3, 4, 5), (3, 23, 22, 54, 2), (98, 2, 12, 26, 3), (4, 6, 9) ]
trends = multi_runs(model1, execution_number=2, iteration_number=100, infection_
↪sets=infection_sets, nprocesses=4)
```

Plot multiple executions

The `ndlib.viz.mpl` package offers support for visualization of multiple runs.

In order to visualize the average trend/prevalence along with its inter-percentile range use the following pattern (assuming `model1` and `trends` be the results of the previous code snippet).

```
from ndlib.viz.mpl.DiffusionTrend import DiffusionTrend
viz = DiffusionTrend(model1, trends)
viz.plot("diffusion.pdf", percentile=90)
```

where `percentile` identifies the upper and lower bound (e.g. setting it to 90 implies a range 10-90).

The same pattern can be also applied to comparison plots.

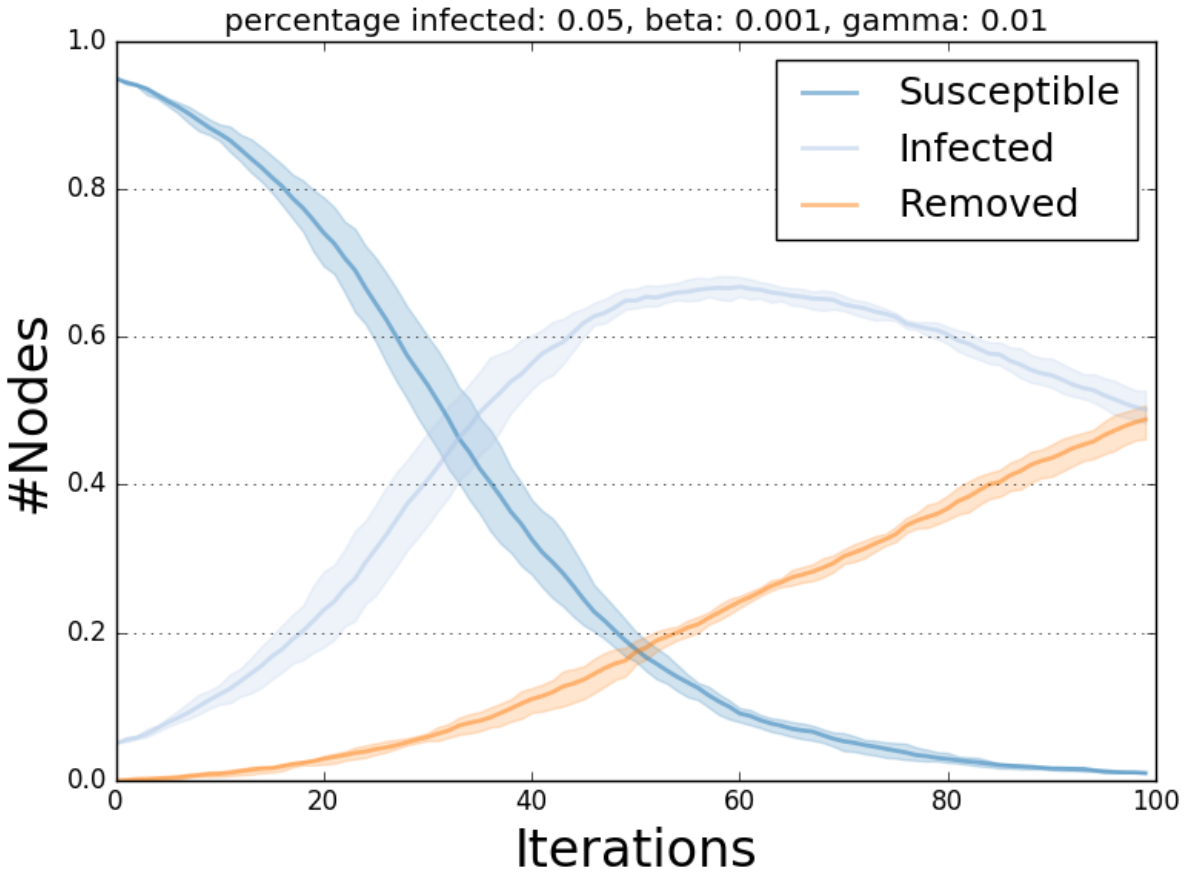

Fig. 1: Multiple run visualization.

1.5.4 Visualization

In order to provide an easy proxy to study diffusion phenomena and compare different configurations as well as models `NDlib` offers built-in visualization facilities.

In particular, the following plots are made available:

Pyplot Viz

Classic Visualizations

Diffusion Trend

The Diffusion Trend plot compares the trends of all the statuses allowed by the diffusive model tested.

Each trend line describes the variation of the number of nodes for a given status iteration after iteration.

class ndlib.viz.mpl.DiffusionTrend.**DiffusionTrend**(*model*, *trends*)

DiffusionTrend.**__init__**(*model*, *trends*)

Parameters

- **model** – The model object
- **trends** – The computed simulation trends

DiffusionTrend.**plot**(*filename*, *percentile*)

Generates the plot

Parameters

- **filename** – Output filename
- **percentile** – The percentile for the trend variance area
- **statuses** – List of statuses to plot. If not specified all statuses trends will be shown.

Below is shown an example of Diffusion Trend description and visualization for the SIR model.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
from ndlib.viz.mpl.DiffusionTrend import DiffusionTrend


# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.001)
cfg.add_model_parameter('gamma', 0.01)
cfg.add_model_parameter("fraction_infected", 0.01)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
trends = model.build_trends(iterations)

# Visualization
viz = DiffusionTrend(model, trends)
viz.plot("diffusion.pdf")
```

Diffusion Prevalence

The Diffusion Prevalence plot compares the delta-trends of all the statuses allowed by the diffusive model tested.

Each trend line describes the delta of the number of nodes for a given status iteration after iteration.

class ndlib.viz.mpl.DiffusionPrevalence.**DiffusionPrevalence**(*model*, *trends*)

DiffusionPrevalence.**__init__**(*model*, *trends*)

Parameters

- **model** – The model object

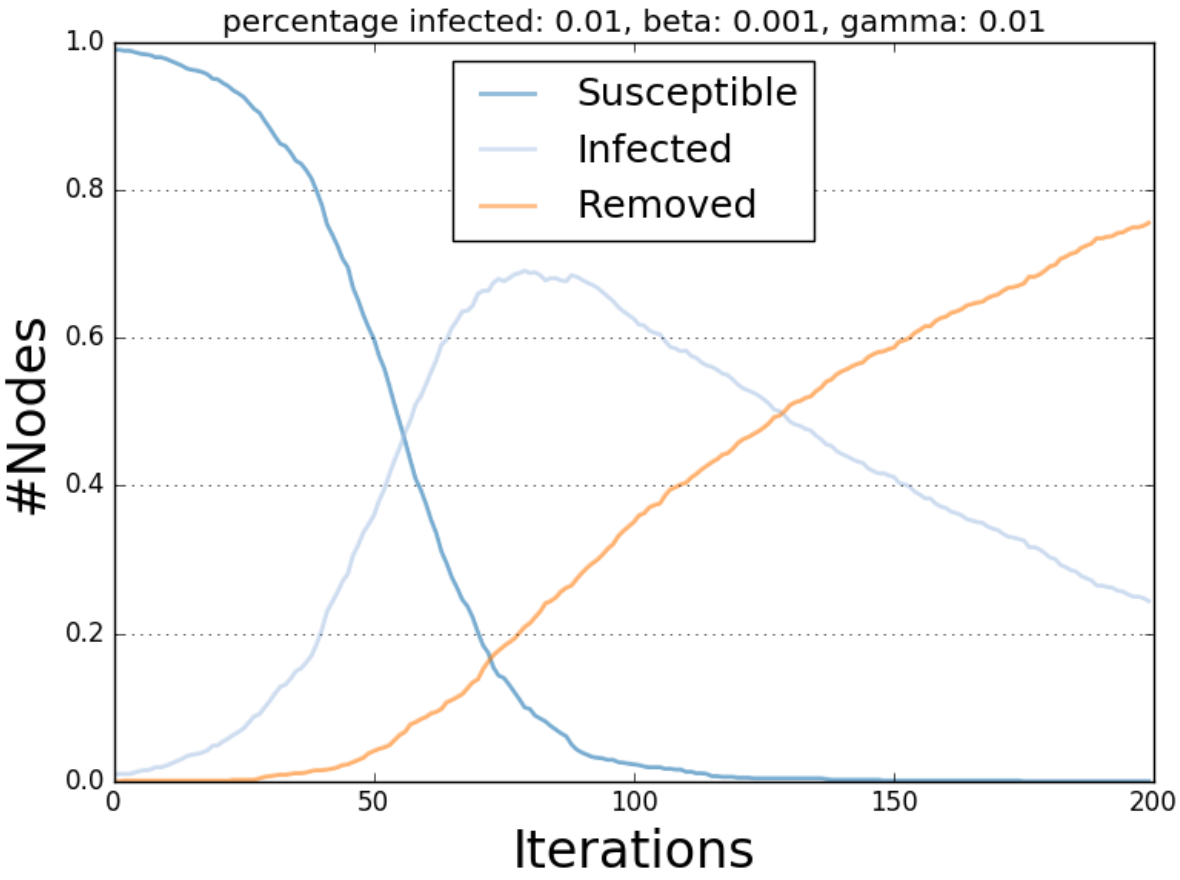

Fig. 2: SIR Diffusion Trend Example.

- **trends** – The computed simulation iterations

DiffusionPrevalence.**plot**(*filename*, *percentile*)

Generates the plot

Parameters

- **filename** – Output filename
- **percentile** – The percentile for the trend variance area
- **statuses** – List of statuses to plot. If not specified all statuses trends will be shown.

Below is shown an example of Diffusion Prevalence description and visualization for the SIR model.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
from ndlib.viz.mpl.DiffusionPrevalence import DiffusionPrevalence


# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.001)
cfg.add_model_parameter('gamma', 0.01)
cfg.add_model_parameter("fraction_infected", 0.01)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
trends = model.build_trends(iterations)

# Visualization
viz = DiffusionPrevalence(model, trends)
viz.plot("prevalence.pdf")
```

Opinion Evolution

The Opinion Evolution plot shows the node-wise opinion evolution in a continuous states model.

class ndlib.viz.mpl.OpinionEvolution.**OpinionEvolution**(*model*, *trends*)

OpinionEvolution.**__init__**(*model*, *trends*)

Parameters

- **model** – The model object
- **trends** – The computed simulation trends

OpinionEvolution.**plot**(*filename*)

Generates the plot

Parameters

- **filename** – Output filename

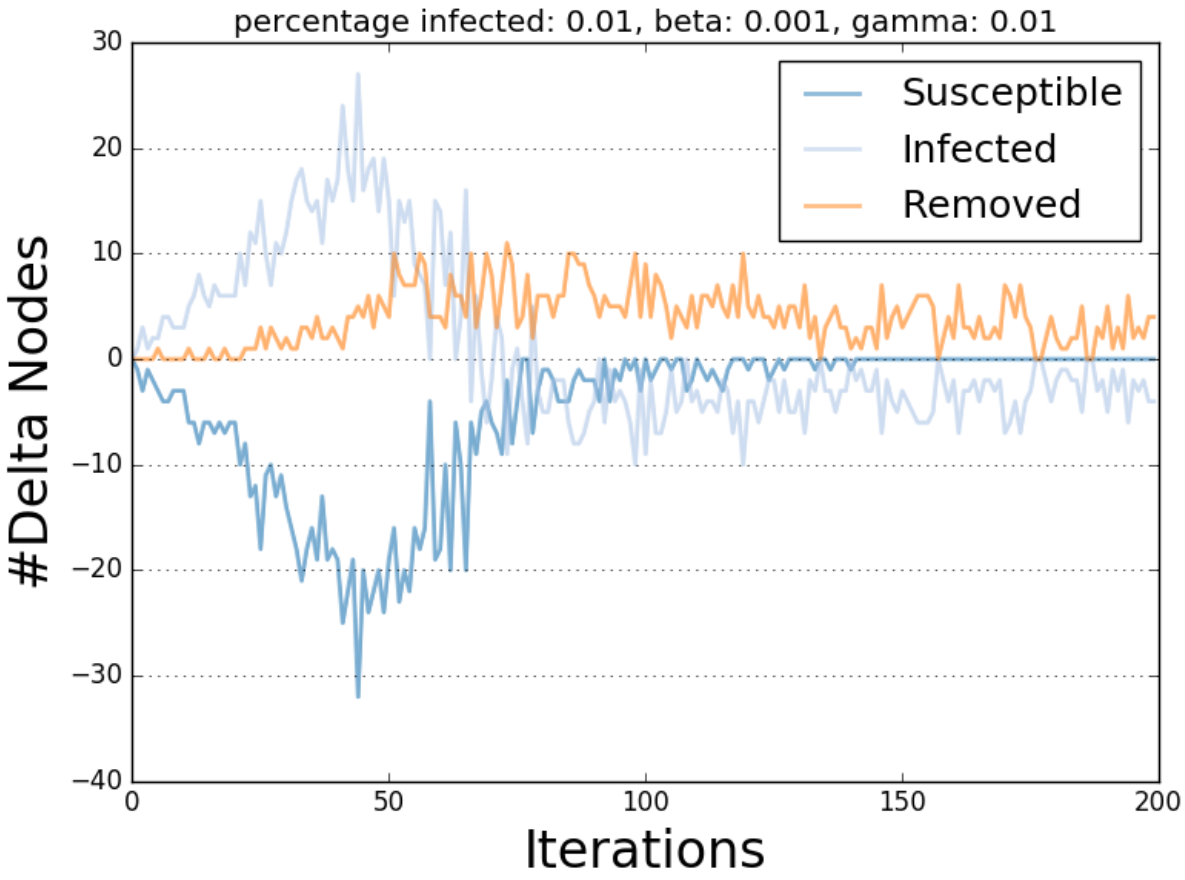

Fig. 3: SIR Diffusion Prevalence Example.

- **percentile** – The percentile for the trend variance area

Below is shown an example of Opinion Evolution description and visualization for the Algorithmic Bias model.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.opinions as op
from ndlib.viz.mpl.OpinionEvolution import OpinionEvolution

# mMean field scenario
g = nx.complete_graph(100)

# Algorithmic Bias model
model = op.AlgorithmicBiasModel(g)

# Model configuration
config = mc.Configuration()
config.add_model_parameter("epsilon", 0.32)
config.add_model_parameter("gamma", 0)
model.set_initial_status(config)

# Simulation execution
iterations = model.iteration_bunch(100)

viz = OpinionEvolution(model, iterations)
viz.plot("opinion_ev.pdf")
```

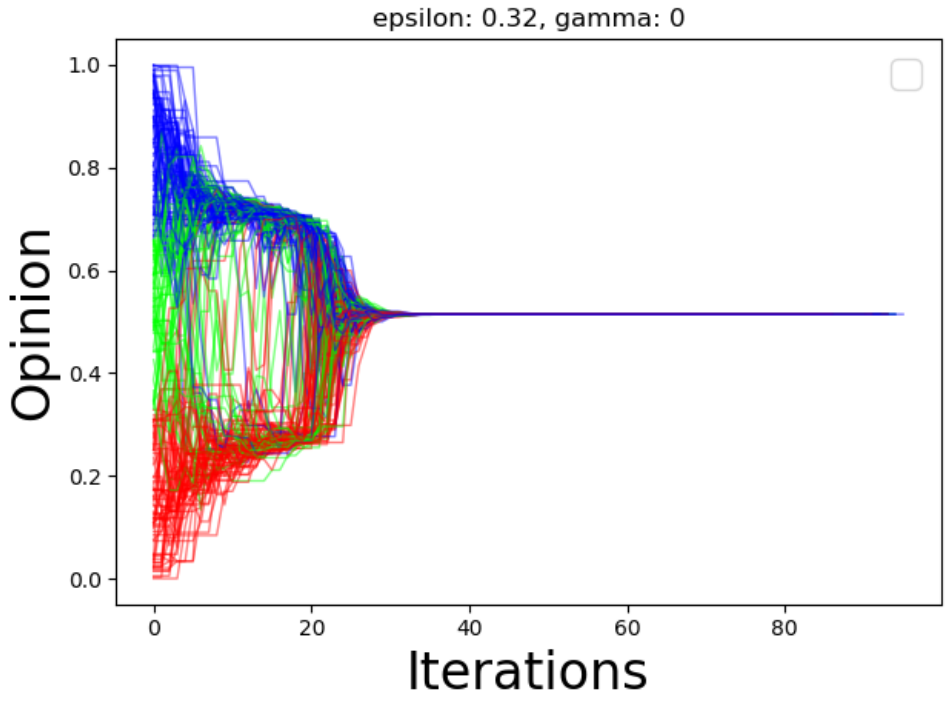

Fig. 4: SIR Diffusion Trend Example.

Model Comparison Visualizations

Diffusion Trend Comparison

The Diffusion Trend Comparison plot compares the trends of all the statuses allowed by the diffusive model tested.

Each trend line describes the variation of the number of nodes for a given status iteration after iteration.

class ndlib.viz.mpl.TrendComparison.**DiffusionTrendComparison**(*models*, *trends*, *statuses='Infected'*)

DiffusionTrendComparison.**__init__**(*models*, *trends*, *statuses*)

Parameters

- **models** – A list of model object
- **trends** – A list of computed simulation trends
- **statuses** – The model statuses for which make the plot. Default ["Infected"].

DiffusionTrendComparison.**plot**(*filename*, *percentile*)

Plot the comparison on file.

Parameters

- **filename** – the output filename
- **percentile** – The percentile for the trend variance area. Default 90.

Below is shown an example of Diffusion Trend description and visualization for the SIR model.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
from ndlib.viz.mpl.TrendComparison import DiffusionTrendComparison


# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.001)
cfg.add_model_parameter('gamma', 0.01)
cfg.add_model_parameter("fraction_infected", 0.01)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
trends = model.build_trends(iterations)

# 2° Model selection
model1 = ep.SIRModel(g)

# 2° Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.001)
cfg.add_model_parameter('gamma', 0.02)
cfg.add_model_parameter("fraction_infected", 0.01)
model1.set_initial_status(cfg)
```

(continues on next page)

(continued from previous page)

```
# 2° Simulation execution
iterations = model1.iteration_bunch(200)
trends1 = model1.build_trends(iterations)

# Visualization
viz = DiffusionTrend([model, model1], [trends, trends1])
viz.plot("trend_comparison.pdf")
```

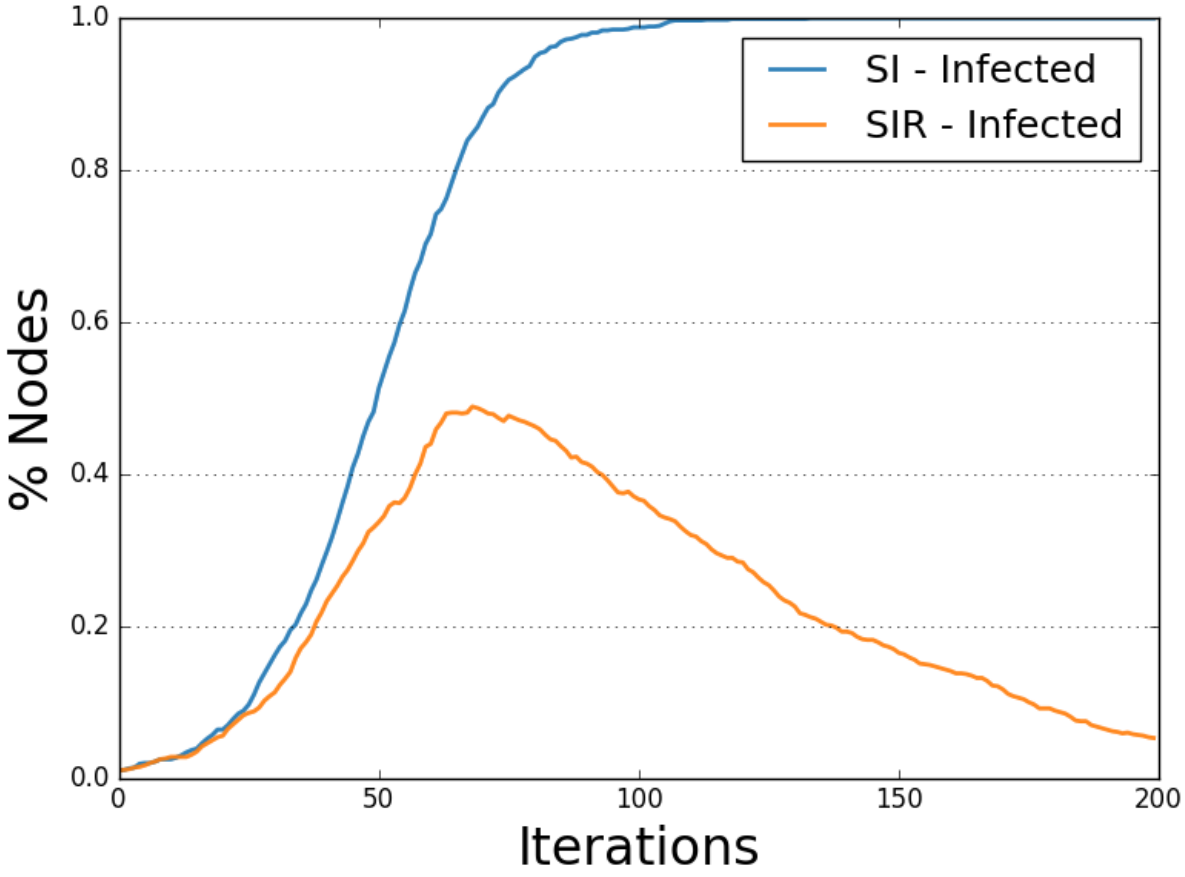

Fig. 5: SIR-SI Diffusion Trend Comparison Example.

Diffusion Prevalence Comparison

The Diffusion Prevalence plot compares the delta-trends of all the statuses allowed by the diffusive model tested.

Each trend line describes the delta of the number of nodes for a given status iteration after iteration.

class ndlib.viz.mpl.PrevalenceComparison.**DiffusionPrevalenceComparison**(*models*, *trends*, *statuses='Infected'*)

DiffusionPrevalenceComparison.**__init__**(*model*, *trends*)

Parameters

- **models** – A list of model object
- **trends** – A list of computed simulation trends
- **statuses** – The model statuses for which make the plot. Default ["Infected"].

DiffusionPrevalenceComparison.**plot**(*filename*, *percentile*)

Plot the comparison on file.

Parameters

- **filename** – the output filename
- **percentile** – The percentile for the trend variance area. Default 90.

Below is shown an example of Diffusion Prevalence description and visualization for two instances of the SIR model.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
from ndlib.viz.mpl.PrevalenceComparison import DiffusionPrevalenceComparison


# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.001)
cfg.add_model_parameter('gamma', 0.02)
cfg.add_model_parameter("fraction_infected", 0.01)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
trends = model.build_trends(iterations)

# 2° Model selection
model1 = ep.SIModel(g)

# 2° Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.001)
cfg.add_model_parameter("fraction_infected", 0.01)
model1.set_initial_status(cfg)

# 2° Simulation execution
iterations = model1.iteration_bunch(200)
trends1 = model1.build_trends(iterations)

# Visualization
viz = DiffusionPrevalenceComparison([model, model1], [trends, trends1])
viz.plot("trend_comparison.pdf")
```

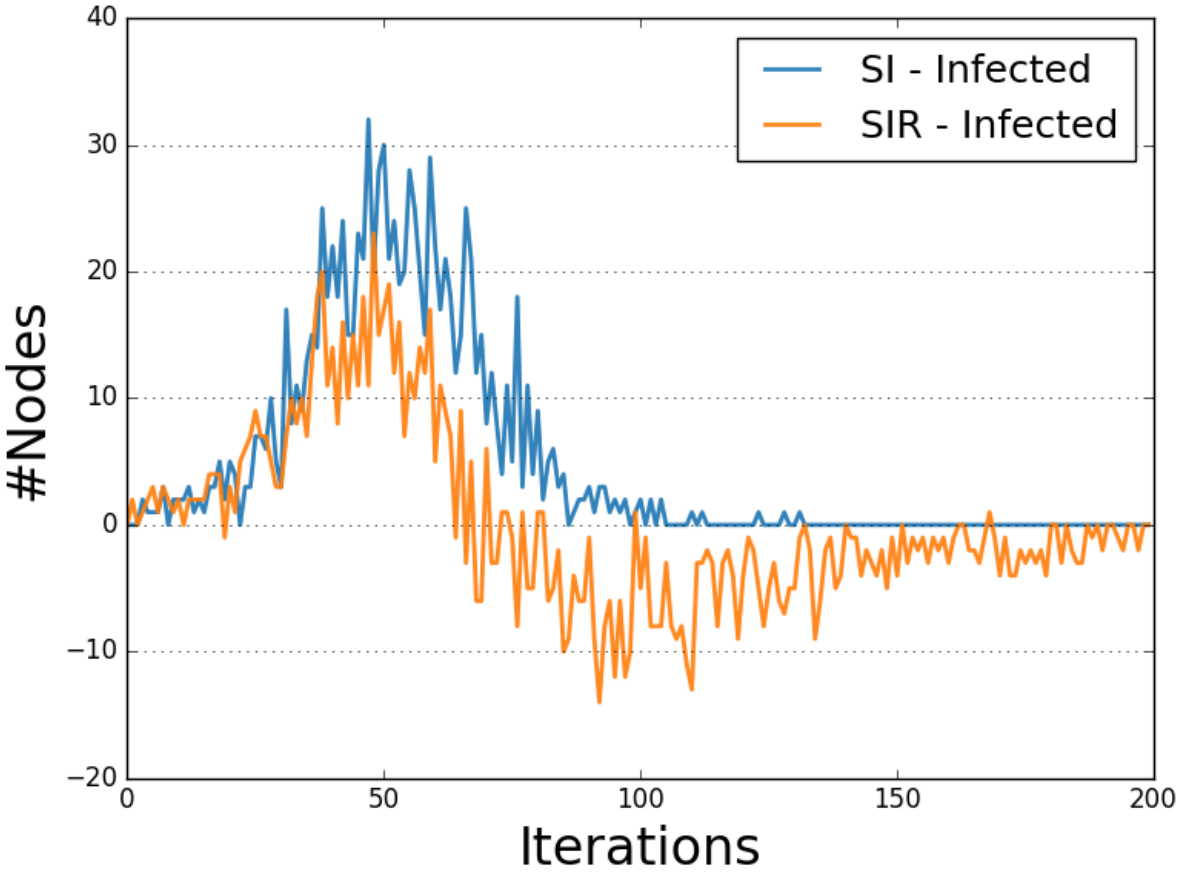

Fig. 6: SIR-SI Diffusion Prevalence Comparison Example.

Bokeh Viz

Classic Visualizations

Diffusion Trend

The Diffusion Trend plot compares the trends of all the statuses allowed by the diffusive model tested.

Each trend line describes the variation of the number of nodes for a given status iteration after iteration.

class ndlib.viz.bokeh.DiffusionTrend.**DiffusionTrend**(*model*, *trends*)

DiffusionTrend.**__init__**(*model*, *iterations*)

Parameters

- **model** – The model object
- **iterations** – The computed simulation iterations

DiffusionTrend.**plot**(*width*, *height*)

Generates the plot

Parameters

- **percentile** – The percentile for the trend variance area
- **width** – Image width. Default 500px.
- **height** – Image height. Default 500px.

Returns a bokeh figure image

Below is shown an example of Diffusion Trend description and visualization for the SIR model.

```
import networkx as nx
from bokeh.io import show
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
from ndlib.viz.bokeh.DiffusionTrend import DiffusionTrend


# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.001)
cfg.add_model_parameter('gamma', 0.01)
cfg.add_model_parameter("fraction_infected", 16 0.05)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
trends = model.build_trends(iterations)

# Visualization
viz = DiffusionTrend(model, trends)
```

(continues on next page)

(continued from previous page)

```
p = viz.plot(width=400, height=400)
show(p)
```

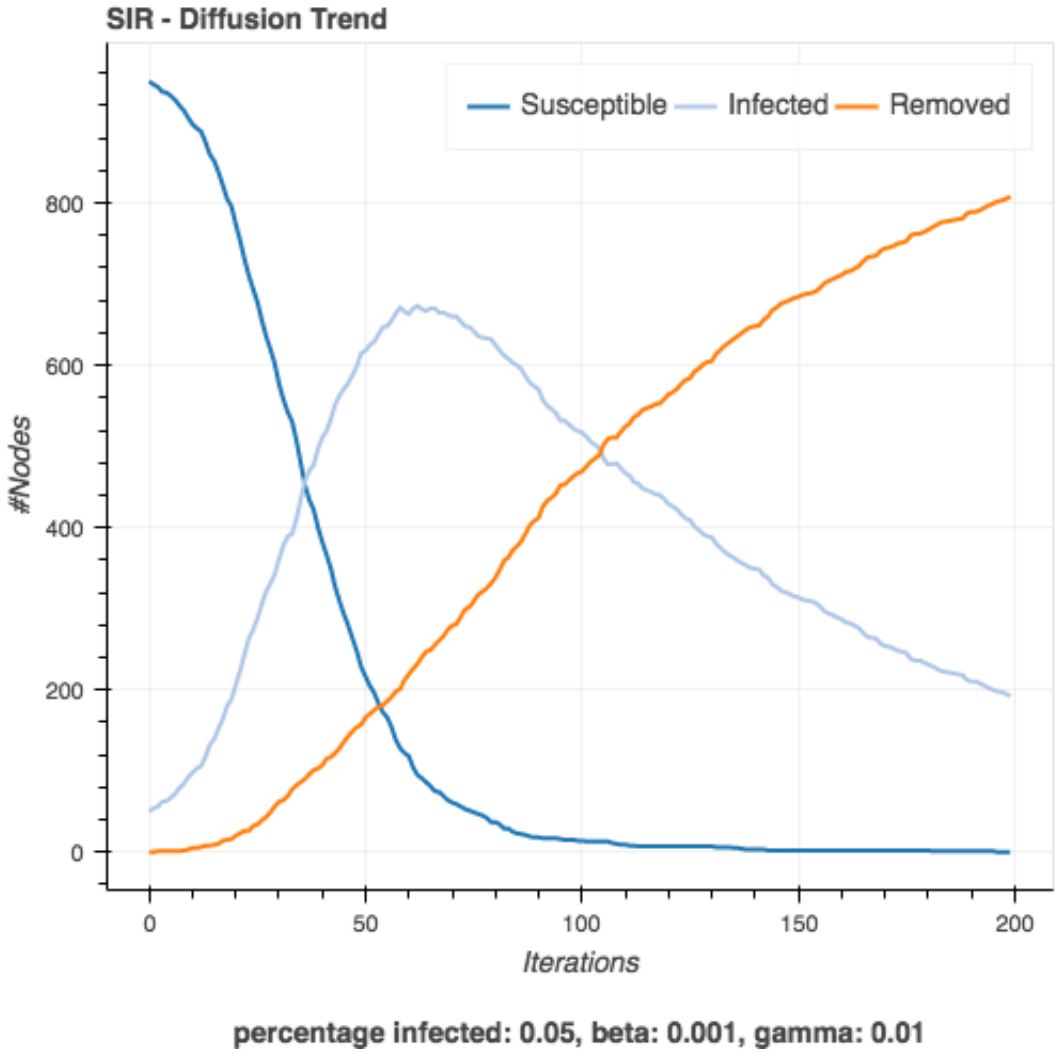

Fig. 7: SIR Diffusion Trend Example.

Diffusion Prevalence

The Diffusion Prevalence plot compares the delta-trends of all the statuses allowed by the diffusive model tested. Each trend line describes the delta of the number of nodes for a given status iteration after iteration.

class ndlib.viz.bokeh.DiffusionPrevalence.**DiffusionPrevalence**(*model*, *trends*)

DiffusionPrevalence.**__init__**(*model*, *iterations*)

Parameters

- **model** – The model object
- **iterations** – The computed simulation iterations

DiffusionPrevalence.**plot**(*width*, *height*)

Generates the plot

Parameters

- **percentile** – The percentile for the trend variance area
- **width** – Image width. Default 500px.
- **height** – Image height. Default 500px.

Returns a bokeh figure image

Below is shown an example of Diffusion Prevalence description and visualization for the SIR model.

```
import networkx as nx
from bokeh.io import show
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
from ndlib.viz.bokeh.DiffusionPrevalence import DiffusionPrevalence


# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.001)
cfg.add_model_parameter('gamma', 0.01)
cfg.add_model_parameter("fraction_infected", 16 0.05)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
trends = model.build_trends(iterations)

# Visualization
viz = DiffusionPrevalence(model, trends)
p = viz.plot(width=400, height=400)
show(p)
```

Model Comparison Visualizations

Multi Plot

The Multi Plot object allows the generation of composite grid figures composed by multiple Diffusion Trends and/or Diffusion Prevalence plots.

class ndlib.viz.bokeh.MultiPlot.**MultiPlot**

MultiPlot.**add_plot**(*plot*)

Parameters **plot** – The bokeh plot to add to the grid

MultiPlot.**plot**(*width*, *height*)

Parameters **ncols** – Number of grid columns

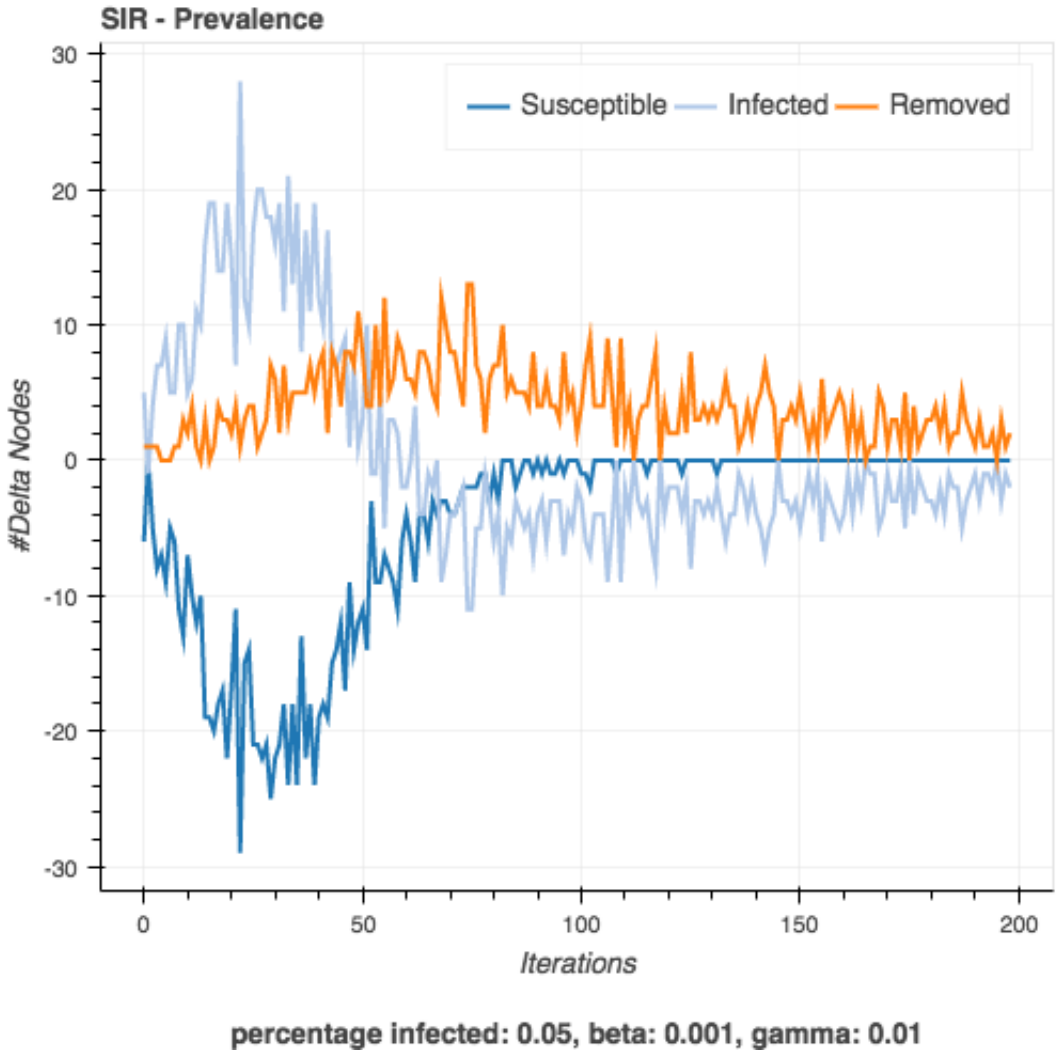

Fig. 8: SIR Diffusion Prevalence Example.

Returns a bokeh figure image

```
import networkx as nx
from bokeh.io import show
import ndlib.models.ModelConfig as mc
import ndlib.models.epidemics as ep
from ndlib.viz.bokeh.DiffusionTrend import DiffusionTrend
from ndlib.viz.bokeh.DiffusionPrevalence import DiffusionPrevalence
from ndlib.viz.bokeh.MultiPlot import Multiplot

vm = MultiPlot()

# Network topology
g = nx.erdos_renyi_graph(1000, 0.1)

# Model selection
model = ep.SIRModel(g)

# Model Configuration
cfg = mc.Configuration()
cfg.add_model_parameter('beta', 0.001)
cfg.add_model_parameter('gamma', 0.01)
cfg.add_model_parameter("fraction_infected", 16 0.05)
model.set_initial_status(cfg)

# Simulation execution
iterations = model.iteration_bunch(200)
trends = model.build_trends(iterations)

# Diffusion Trend
viz = DiffusionTrend(model, trends)
p = viz.plot(width=400, height=400)
vm.add_plot(p)

# Diffusion Prevalence
viz = DiffusionPrevalence(model, trends)
p1 = viz.plot(width=400, height=400)

vm.add_plot(p1)

m = vm.plot(ncol=2)
show(m)
```

1.6 Custom Model Definition

NDlib exposes a set of built-in diffusion models (epidemic/opinion dynamics/dynamic network): how can I describe novel ones?

In order to answer such question we developed a syntax for compositional model definition.

1.6.1 Rationale

At a higher level of abstraction a diffusion process can be synthesized into two components:

- Available Statuses, and

- Transition Rules that connect them

All models of `NDlib` assume an agent-based, discrete time, simulation engine. During each simulation iteration all the nodes in the network are asked to (i) evaluate their current status and to (ii) (eventually) apply a matching transition rule. The last step of such process can be easily decomposed into atomic operations that we will call *compartments*.

Note: `NDlib` exposes two classes for defining custom diffusion models:

- `CompositeModel` describes diffusion models for static networks
- `DynamicCompositeModel` describes diffusion models for dynamic networks

To avoid redundant documentation, here we will discuss only the former class, the latter behaving alike.

1.6.2 Compartments

We adopt the concept of `compartment` to identify all those atomic conditions (i.e. operations) that describe (part of) a transition rule. The execution of a `compartment` can return either *True* (condition satisfied) or *False* (condition not satisfied).

Indeed, several compartments can be described, each one of them capturing an atomic operation.

To cover the main scenarios we defined three families of compartments as well as some operations to combine them.

Node Compartments

In this class fall all those compartments that evaluate conditions tied to **node** status/features. They model stochastic events as well as deterministic ones.

Node Stochastic

Node Stochastic compartments are used to evaluate stochastic events attached to network nodes.

Consider the transition rule **Susceptible->Infected** that requires a probability *beta* to be satisfied. Such rule can be described by a simple compartment that models Node Stochastic behaviors. Let's call il *NS*.

The rule will take as input the *initial* node status (Susceptible), the *final* one (Infected) and the *NS* compartment. *NS* will thus require a probability (*beta*) of activation.

During each rule evaluation, given a node *n*

- **if the actual status of *n* equals the rule *initial* one**
  - a random value *b* in [0,1] will be generated
  - if *b* <= *beta* then *NS* is considered *satisfied* and the status of *n* changes from *initial* to *final*.

Moreover, *NS* allows to specify a *triggering* status in order to restrain the compartment evaluation to those nodes that:

1. match the rule *initial* state, and
2. have at least one neighbors in the *triggering* status.

Parameters

Name	Value Type	Default	Mandatory	Description
ratio	float in [0, 1]		True	Event probability
triggering_status	string	None	False	Trigger

Example

In the code below is shown the formulation of a SIR model using NodeStochastic compartments.

The first compartment, *c1*, is used to implement the transition rule *Susceptible->Infected*. It requires a probability threshold - here set equals to 0.02 - and restrain the rule evaluation to all those nodes that have at least an Infected neighbors.

The second compartment, *c2*, is used to implement the transition rule *Infected->Removed*. Since such transition is not tied to neighbors statuses the only parameter required by the compartment is the probability of transition.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.NodeStochastic as ns

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")
model.add_status("Removed")

# Compartment definition
c1 = ns.NodeStochastic(0.02, triggering_status="Infected")
c2 = ns.NodeStochastic(0.01)

# Rule definition
model.add_rule("Susceptible", "Infected", c1)
model.add_rule("Infected", "Removed", c2)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

Node Categorical Attribute

Node Categorical Attribute compartments are used to evaluate events attached to network nodes attributes.

Consider the transition rule **Susceptible->Infected** that requires a that the susceptible node express a specific value of an internal attribute, *attr*, to be satisfied (e.g. "Sex"="male"). Such rule can be described by a simple compartment that models Node Categorical Attribute selection. Let's call il *NCA*.

The rule will take as input the *initial* node status (Susceptible), the *final* one (Infected) and the *NCA* compartment. *NCA* will thus require a probability (*beta*) of activation.

During each rule evaluation, given a node *n*

- **if the actual status of *n* equals the rule *initial* one**
  - a random value *b* in [0,1] will be generated
  - if *b* <= *beta* and *attr(n)* == *attr*, then *NCA* is considered *satisfied* and the status of *n* changes from *initial* to *final*.

Parameters

Name	Value Type	Default	Mandatory	Description
attribute	string	None	True	Attribute name
value	string	None	True	Attribute testing value
probability	float in [0, 1]	1	False	Event probability

Example

In the code below is shown the formulation of a model using NodeCategoricalAttribute compartments.

The compartment, *c1*, is used to implement the transition rule *Susceptible->Infected*. It restrain the rule evaluation to all those nodes for which the attribute "Sex" equals "male".

```
import networkx as nx
import random
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.NodeCategoricalAttribute as ns

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Setting node attribute
attr = {n: {"Sex": random.choice(['male', 'female'])} for n in g.nodes()}
nx.set_node_attributes(g, attr)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")
model.add_status("Removed")

# Compartment definition
c1 = na.NodeCategoricalAttribute("Sex", "male", probability=0.6)

# Rule definition
```

(continues on next page)

(continued from previous page)

```
model.add_rule("Susceptible", "Infected", c1)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

Node Numerical Attribute

Node Numerical Attribute compartments are used to evaluate events attached to numeric edge attributes.

Consider the transition rule **Susceptible->Infected** that requires a that the susceptible node express a specific value of an internal numeric attribute, *attr*, to be satisfied (e.g. "Age" == 18). Such rule can be described by a simple compartment that models Node Numerical Attribute selection. Let's call il *NNA*.

The rule will take as input the *initial* node status (Susceptible), the *final* one (Infected) and the *NNA* compartment. *NNA* will thus require a probability (*beta*) of activation.

During each rule evaluation, given a node *n* and one of its neighbors *m*

- **if the actual status of *n* equals the rule *initial***
  - if *attr(n)* **op** *attr*
  - a random value *b* in [0,1] will be generated
  - if *b* <= *beta*, then *NNA* is considered *satisfied* and the status of *n* changes from *initial* to *final*.

op represent a logic operator and can assume one of the following values: - equality: "==" - less than: "<" - greater than: ">" - equal or less than: "<=" - equal or greater than: ">=" - not equal to: "!=" - within: "IN"

Moreover, *NNA* allows to specify a *triggering* status in order to restrain the compartment evaluation to those nodes that:

1. match the rule *initial* state, and
2. have at least one neighbors in the *triggering* status.

Parameters

Name	Value Type	Default	Mandatory	Description
attribute	string	None	True	Attribute name
value	numeric(*)	None	True	Attribute testing value
op	string	None	True	Logic operator
probability	float in [0, 1]	1	False	Event probability
triggering_status	string	None	False	Trigger

(*) When *op* equals "IN" the attribute *value* is expected to be a tuple of two elements identifying a closed interval.

Example

In the code below is shown the formulation of a model using NodeNumericalAttribute compartments.

The first compartment, *c1*, is used to implement the transition rule *Susceptible->Infected*. It restrain the rule evaluation to all those nodes having "Age" equals to 18.

The second compartment, *c2*, is used to implement the transition rule *Infected->Recovered*. It restrain the rule evaluation to all those nodes connected at least to a "Susceptible" neighbor and having "Age" in the range [20, 25].

```
import networkx as nx
import random
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.NodeNumericalAttribute as na

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Setting edge attribute
attr = {n: {"Age": random.choice(range(0, 100))} for n in g.nodes()}
nx.set_node_attributes(g, attr)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")
model.add_status("Removed")

# Compartment definition
c1 = na.NodeNumericalAttribute("Age", value=18, op="==", probability=0.6)
c2 = na.NodeNumericalAttribute("Age", value=[20, 25], op="IN", probability=0.6,␣
↪triggering_status="Susceptible")

# Rule definition
model.add_rule("Susceptible", "Infected", c1)
model.add_rule("Infected", "Removed", c2)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

Node Threshold

Node Threshold compartments are used to evaluate deterministic events attached to network nodes.

Consider the transition rule **Susceptible->Infected** that requires at least a percentage *beta* of *Infected* neighbors for a node *n* to be satisfied.

Such rule can be described by a simple compartment that models Node Threshold behaviors. Let's call il *NT*.

The rule will take as input the *initial* node status (Susceptible), the *final* one (Infected) and the *NT* compartment. *NT* will thus require a threshold (*beta*) of activation and a *triggering* status.

During each rule evaluation, given a node *n*

- **if the actual status of *n* equals the rule *initial* one**
  - let *b* identify the ratio of *n* neighbors in the *triggering* status
  - if *b* >= *beta* then *NS* is considered *satisfied* and the status of *n* changes from *initial* to *final*.

Parameters

Name	Value Type	Default	Mandatory	Description
threshold	float in [0, 1]		False	Node threshold
triggering_status	string	None	True	Trigger

Example

In the code below is shown the formulation of a Threshold model using NodeThreshold compartments.

The compartment, *c1*, is used to implement the transition rule *Susceptible->Infected*. It requires a threshold - here set equals to 0.2.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.NodeThreshold as ns

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")

# Compartment definition
c1 = ns.NodeThreshold(0.1, triggering_status="Infected")

# Rule definition
model.add_rule("Susceptible", "Infected", c1)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

In case of an heterogeneous node threshold distribution the same model can be expressed as follows

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.NodeThreshold as ns

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")

# Compartment definition
c1 = ns.NodeThreshold(triggering_status="Infected")

# Rule definition
model.add_rule("Susceptible", "Infected", c1)

# Model initial status configuration
config = mc.Configuration()

# Threshold specs
for i in g.nodes():
        config.add_node_configuration("threshold", i, np.random.random_sample())

config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

Edge Compartments

In this class fall all those compartments that evaluate conditions tied to **edge** features. They model stochastic events as well as deterministic ones.

Edge Stochastic

Edge Stochastic compartments are used to evaluate stochastic events attached to network edges.

Consider the transition rule **Susceptible->Infected** that, to be triggered, requires a direct link among an infected node and a susceptible one. Moreover, it can happens subject to probability *beta*, a parameter tied to the specific edge connecting the two nodes. Such rule can be described by a simple compartment that models Edge Stochastic behaviors. Let's call il *ES*.

The rule will take as input the *initial* node status (Susceptible), the *final* one (Infected) and the *ES* compartment. *ES* will thus require a probability (*beta*) of edge activation and a *triggering* status. In advanced scenarios, where the probability threshold vary from edge to edge, it is possible to specify it using the model configuration object.

During each rule evaluation, given a node *n* and one of its neighbors *m*

- **if the actual status of *n* equals the rule *initial* one and the one of *m* equals the *triggering* one**

- a random value *b* in [0,1] will be generated
- if *b* <= *beta* then *ES* is considered *satisfied* and the status of *n* changes from *initial* to *final*.

Parameters

Name	Value Type	Default	Mandatory	Description
threshold	float in [0, 1]	1/N	True	Event probability
triggering_status	string	None	False	Trigger

Where N is the number of nodes in the graph.

Example

In the code below is shown the formulation of a Cascade model using EdgeStochastic compartments.

The compartment, *c1*, is used to implement the transition rule *Susceptible->Infected*. It requires a probability threshold - here set equals to 0.02 - and restrain the rule evaluation to all those nodes that have at least an Infected neighbors.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.EdgeStochastic as es

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")
model.add_status("Removed")

# Compartment definition
c1 = ns.EdgeStochastic(0.02, triggering_status="Infected")

# Rule definition
model.add_rule("Susceptible", "Infected", c1)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

In case of an heterogeneous edge threshold distribution the same model can be expressed as follows

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
```

(continues on next page)

(continued from previous page)

```
import ndlib.models.compartments.EdgeStochastic as es

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")
model.add_status("Removed")

# Compartment definition
c1 = es.EdgeStochastic(triggering_status="Infected")

# Rule definition
model.add_rule("Susceptible", "Infected", c1)


# Model initial status configuration
config = mc.Configuration()

# Threshold specs
for e in g.edges():
        config.add_edge_configuration("threshold", e, np.random.random_sample())

config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

Edge Categorical Attribute

Edge Categorical Attribute compartments are used to evaluate events attached to edge attributes.

Consider the transition rule **Susceptible->Infected** that requires a that the susceptible node is connected to a neighbor through a link expressing a specific value of an internal attribute, *attr*, to be satisfied (e.g. "type"="co-worker"). Such rule can be described by a simple compartment that models Edge Categorical Attribute selection. Let's call il *ECA*.

The rule will take as input the *initial* node status (Susceptible), the *final* one (Infected) and the *ECA* compartment. *ECA* will thus require a probability (*beta*) of activation.

During each rule evaluation, given a node *n* and one of its neighbors *m*

- **if the actual status of *n* equals the rule *initial***
  - if *attr(n,m) == attr*
  - a random value *b* in [0,1] will be generated
  - if *b* <= *beta*, then *ECA* is considered *satisfied* and the status of *n* changes from *initial* to *final*.

Moreover, *ECA* allows to specify a *triggering* status in order to restrain the compartment evaluation to those nodes that:

1. match the rule *initial* state, and
2. have at least one neighbors in the *triggering* status.

Parameters

Name	Value Type	Default	Mandatory	Description
attribute	string	None	True	Attribute name
value	string	None	True	Attribute testing value
probability	float in [0, 1]	1	False	Event probability
triggering_status	string	None	False	Trigger

Example

In the code below is shown the formulation of a model using EdgeCategoricalAttribute compartments.

The first compartment, *c1*, is used to implement the transition rule *Susceptible->Infected*. It restrain the rule evaluation to all those nodes connected through a link having the attribute "type" equals "co-worker".

The second compartment, *c2*, is used to implement the transition rule *Infected->Recovered*. It restrain the rule evaluation to all those nodes connected trough a link having the attribute "type" equals "family" whose neighbors is "Susceptible".

```
import networkx as nx
import random
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.EdgeCategoricalAttribute as ns

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Setting edge attribute
attr = {e: {"type": random.choice(['co-worker', 'family'])} for e in g.edges()}
nx.set_edge_attributes(g, attr)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")
model.add_status("Removed")

# Compartment definition
c1 = na.NodeCategoricalAttribute("type", "co-worker", probability=0.6)
c2 = na.NodeCategoricalAttribute("type", "family", probability=0.6, triggering_status=
↪"Susceptible")

# Rule definition
model.add_rule("Susceptible", "Infected", c1)
model.add_rule("Infected", "Recovered", c2)

# Model initial status configuration
```

(continues on next page)

(continued from previous page)

```
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

Edge Numerical Attribute

Edge Numerical Attribute compartments are used to evaluate events attached to numeric edge attributes.

Consider the transition rule **Susceptible->Infected** that requires a that the susceptible node is connected to a neighbor through a link expressing a specific value of an internal numeric attribute, *attr*, to be satisfied (e.g. "weight">=3). Such rule can be described by a simple compartment that models Edge Numerical Attribute selection. Let's call il *ENA*.

The rule will take as input the *initial* node status (Susceptible), the *final* one (Infected) and the *ENA* compartment. *ENA* will thus require a probability (*beta*) of activation.

During each rule evaluation, given a node *n* and one of its neighbors *m*

- **if the actual status of *n* equals the rule *initial***
  - if *attr(n,m)* **op** *attr*
  - a random value *b* in [0,1] will be generated
  - if *b* <= *beta*, then *ECA* is considered *satisfied* and the status of *n* changes from *initial* to *final*.

op represent a logic operator and can assume one of the following values: - equality: "==" - less than: "<" - greater than: ">" - equal or less than: "<=" - equal or greater than: ">=" - not equal to: "!=" - within: "IN"

Moreover, *ENA* allows to specify a *triggering* status in order to restrain the compartment evaluation to those nodes that:

1. match the rule *initial* state, and
2. have at least one neighbors in the *triggering* status.

Parameters

Name	Value Type	Default	Mandatory	Description
attribute	string	None	True	Attribute name
value	numeric(*)	None	True	Attribute testing value
op	string	None	True	Logic operator
probability	float in [0, 1]	1	False	Event probability
triggering_status	string	None	False	Trigger

(*) When *op* equals "IN" the attribute *value* is expected to be a tuple of two elements identifying a closed interval.

Example

In the code below is shown the formulation of a model using EdgeNumericalAttribute compartments.

The first compartment, *c1*, is used to implement the transition rule *Susceptible->Infected*. It restrain the rule evaluation to all those nodes connected at least to a neighbor through a link having "weight" equals to 4.

The second compartment, *c2*, is used to implement the transition rule *Infected->Recovered*. It restrain the rule evaluation to all those nodes connected at least to a "Susceptible" neighbor through a link having "weight" in the range [3, 6].

```
import networkx as nx
import random
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.EdgeNumericalAttribute as ns

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Setting edge attribute
attr = {(u, v): {"weight": int((u+v) % 10)} for (u, v) in g.edges()}
nx.set_edge_attributes(g, attr)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")
model.add_status("Removed")

# Compartment definition
c1 = na.EdgeNumericalAttribute("weight", value=4, op="==", probability=0.6)
c2 = na.EdgeNumericalAttribute("weight", value=(3, 6), op="IN", probability=0.6,␣
↪triggering_status="Susceptible")

# Rule definition
model.add_rule("Susceptible", "Infected", c1)
model.add_rule("Infected", "Recovered", c2)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

Time Compartments

In this class fall all those compartments that evaluate conditions tied to **temporal execution**. They can be used to model, for instance, lagged events as well as triggered transitions.

Count Down

Count Down compartments are used to evaluate time related deterministic events attached to network nodes.

Consider the transition rule **Susceptible->Infected** that has an latent period of *t* iterations.

Such rule can be described by a simple compartment that models Count Down behaviors. Let's call il *CD*.

The rule will take as input the *initial* node status (Susceptible), the *final* one (Infected) and the *CD* compartment. *CD* will thus require a countdown name (*cn*) and the number of iterations (*t*) before activation.

During each rule evaluation, given a node *n*

- **if the actual status of *n* equals the rule *initial* one**
  - if the node does not have an associated countdown *cn* initialize it to *t*
  - **else**
    - if *cn(t)* > *t* decrement *cn(t)*
    - if *cn(t)* <= *t* then *CD* is considered *satisfied* and the status of *n* changes from *initial* to *final*.

Parameters

Name	Value Type	Default	Mandatory	Description
name	string	None	True	Count Down name
iterations	int	None	True	Duration

Example

In the code below is shown the formulation of a model using CountDown compartments.

The compartment, *c1*, is used to implement the transition rule *Susceptible->Infected*. It requires activates after 10 iteration.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.CountDown as cd

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")

# Compartment definition
c1 = cd.CountDown("incubation", iterations=10)

# Rule definition
model.add_rule("Susceptible", "Infected", c1)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

1.6.3 Compartments Composition

Compartment can be chained in multiple ways so to describe complex transition rules. In particular, a transition rule can be seen as a tree whose nodes are compartments and edges connections among them.

- The initial node status is evaluated at the root of the tree (the *master* compartment)
- if the operation described by such compartment is satisfied the conditions of (one of) its child compartments is evaluated
- if a path from the root to one leaf of the tree is completely satisfied the transition rule applies and the node change its status.

Compartments can be combined following two criteria:

Cascading Composition

Since each compartment identifies an atomic condition it is natural to imagine rules described as *chains* of compartments.

A compartment chain identify and ordered set of conditions that needs to be satisfied to allow status transition (it allows describing an **AND** logic).

To implement such behaviour each compartment exposes a parameter (named *composed*) that allows to specify the subsequent compartment to evaluate in case it condition is satisfied.

Example

In the code below is shown the formulation of a model implementing cascading compartment composition.

The rule **Susceptible->Infected** is implemented using three NodeStochastic compartments chained as follows:

- **If the node *n* is *Susceptible***
  - *c1*: if at least a neighbor of the actual node is *Infected*, with probability 0.5 evaluate compartment *c2*
  - *c2*: with probability 0.4 evaluate compartment *c3*
  - *c3*: with probability 0.2 allow the transition to the *Infected* state

Indeed, heterogeneous compartment types can be mixed to build more complex scenarios.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.NodeStochastic as ns

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")

# Compartment definition and chain construction
c3 = ns.NodeStochastic(0.2)
```

(continues on next page)

(continued from previous page)

```
c2 = ns.NodeStochastic(0.4, composed=c3)
c1 = ns.NodeStochastic(0.5, "Infected", composed=c2)

# Rule definition
model.add_rule("Susceptible", "Infected", c1)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

Conditional Composition

Since each compartment identifies an atomic condition it is natural to imagine rules described as *trees* of compartments.

A compartment tree identify and ordered and disjoint set of conditions that needs to be satisfied to allow status transition (it allows describing an **OR** logic).

To implement such behaviour we implemented a ConditionalComposition compartment that allows to describe branching. Let's call it *CC*.

CC evaluate a *guard* compartment and, depending from the result it gets evaluate (True or False) move to the evaluation of one of its two *child* compartments.

Parameters

Name	Value Type	Default	Mandatory	Description
condition	Compartment	None	True	Guard Compartment
first_branch	Compartment	None	True	Positive Compartment
second_branch	Compartment	None	True	Negative Compartment

Example

In the code below is shown the formulation of a model implementing conditional compartment composition.

The rule **Susceptible->Infected** is implemented using three NodeStochastic compartments chained as follows:

- **If the node *n* is *Susceptible***
  - *c1*: if at least a neighbor of the actual node is *Infected*, with probability 0.5 evaluate compartment *c2* else evaluate compartment *c3*
  - *c2*: with probability 0.2 allow the transition to the *Infected* state
  - *c3*: with probability 0.1 allow the transition to the *Infected* state

Indeed, heterogeneous compartment types can be mixed to build more complex scenarios.

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.NodeStochastic as ns
import ndlib.models.compartments.ConditionalComposition as cif

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
model.add_status("Susceptible")
model.add_status("Infected")

# Compartment definition
c1 = ns.NodeStochastic(0.5, "Infected")
c2 = ns.NodeStochastic(0.2)
c3 = ns.NodeStochastic(0.1)

# Conditional Composition
cc = cif.ConditionalComposition(c1, c2, c3)

# Rule definition
model.add_rule("Susceptible", "Infected", cc)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(100)
```

A rule can be defined by employing all possible combinations of cascading and conditional compartment composition.

1.6.4 Examples

Here some example of models implemented using compartments.

SIR

```
import networkx as nx
import ndlib.models.ModelConfig as mc
import ndlib.models.CompositeModel as gc
import ndlib.models.compartments.NodeStochastic as ns

# Network generation
g = nx.erdos_renyi_graph(1000, 0.1)

# Composite Model instantiation
model = gc.CompositeModel(g)

# Model statuses
```

(continues on next page)

(continued from previous page)

```
model.add_status("Susceptible")
model.add_status("Infected")
model.add_status("Removed")

# Compartment definition
c1 = ns.NodeStochastic(0.02, triggering_status="Infected")
c2 = ns.NodeStochastic(0.01)

# Rule definition
model.add_rule("Susceptible", "Infected", c1)
model.add_rule("Infected", "Removed", c2)

# Model initial status configuration
config = mc.Configuration()
config.add_model_parameter('fraction_infected', 0.1)

# Simulation execution
model.set_initial_status(config)
iterations = model.iteration_bunch(5)
```

1.7 NDQL: Network Diffusion Query Language

NDlib aims to an heterogeneous audience composed by technicians as well as analysts. In order to abstract from the its programming interface we designed a query language to describe diffusion simulations, NDQL.

1.7.1 Rationale

NDQL is built upon the custom model definition facilities offered by NDlib.

It provides a simple, declarative, syntax for describing and executing diffusion simulations by

- **creating a custom model composed of**
  - node statuses;
  - transition rules (expressed as combinations of compartments)
- creating a synthetic graph / loading an existing network
- initialize initial nodes statuses
- run the simulation

NDQL is designed to allow those users that are not familiar to the Python language to:

- abstract the technicality of the programming interface, and
- directly describe the expected model behaviour

So far, NDQL supports only static network analysis.

1.7.2 NDQL Syntax

An NDQL script is composed of a minimum set of directives:

- **Model definition:**

  - MODEL, STATUS, COMPARTMENT (+), IF-THEN-ELSE (+), RULE,
- **Model initialization:**
  - INITIALIZE
- **Network specification:**
  - CREATE_NETWORK ($), LOAD_NETWORK ($)
- **Simulation execution:**
  - EXECUTE

Directives marked with (+) are optional while the ones marked with ($) are mutually exclusive w.r.t. their class.

The complete language directive specification is the following:

```
MODEL model_name

STATUS status_name

COMPARTMENT compartment_name
TYPE compartment_type
COMPOSE compartment_name
[PARAM param_name numeric]*
[TRIGGER status_name]

IF compartment_name_1 THEN compartment_name_2 ELSE compartment_name_3 AS rule_name

RULE rule_name
FROM status_name
TO status_name
USING compartment_name

INITIALIZE
[SET status_name ratio]+

CREATE_NETWORK network_name
TYPE network_type
[PARAM param_name numeric]*

LOAD_NETWORK network_name FROM network_file

EXECUTE model_name ON network_name FOR iterations
```

The CREATE_NETWORK directive can take as *network_type* any `networkx` graph generator name (*param_name* are inherited from generator function parameters).

1.7.3 Execute/Translate NDQL files

`NDlib` installs two command line commands: - `NDQL_translate` - `NDQL_execute`

The former command allows to translate a generic, well-formed, `NDQL` script into an equivalent Python one. It can be executed as

```
NDQL_translate query_file python_file
```

where *query_file* identifies the target `NDQL` script and *python_file* specifies the desired name for the resulting Python script.

The latter command allows to directly execute a generic, well-formed, NDQL script.It can be executed as

```
NDQL_execute query_file result_file
```

where *query_file* identifies the target NDQL script and *result_file* specifies the desired name for the execution results. Execution results are saved as JSON files with the following syntax:

```
[{"trends":
   {
    "node_count": {"0": [270, 179, 15, 0, 0], "1": [30, 116, 273, 256, 239], "2": [0,␣
↪5, 12, 44, 61]},
    "status_delta": {"0": [0, -91, -164, -15, 0], "1": [0, 86, 157, -17, -17], "2":␣
↪[0, 5, 7, 32, 17]}
    },
    "Statuses": {"1": "Infected", "2": "Removed", "0": "Susceptible"}
 }]
```

where - *node_count* describe the trends built on the number of nodes per status - *status_delta* describe the trends built on the fluctuations of number of nodes per status - *Statuses* provides a map from numerical id to status name

1.7.4 Examples

Here some example of models implemented using NDQL.

SIR

```
CREATE_NETWORK g1
TYPE erdos_renyi_graph
PARAM n 300
PARAM p 0.1

MODEL SIR

STATUS Susceptible
STATUS Infected
STATUS Removed

# Compartment definitions

COMPARTMENT c1
TYPE NodeStochastic
PARAM rate 0.1
TRIGGER Infected

COMPARTMENT c2
TYPE NodeStochastic
PARAM rate 0.1

# Rule definitions

RULE
FROM Susceptible
TO Infected
USING c1
```

(continues on next page)

(continued from previous page)

```
RULE
FROM Infected
TO Removed
USING c2

# Model configuration

INITIALIZE
SET Infected 0.1

EXECUTE SIR ON g1 FOR 5
```

1.8 Experiment Server

The simulation facilities offered by `NDlib` are specifically designed for those users that want to run experiments on their local machine.

However, in some scenarios, e.g. due to limited computational resources or to the rising of other particular needs, it may be convenient to separate the machine on which the definition of the experiment is made from the one that actually executes the simulation.

In order to satisfy such needs, we developed a RESTfull service, `NDlib-REST`, that builds upon `NDlib` an experiment server queryable through API calls.

Project Website: https://github.com/GiulioRossetti/ndlib-rest

1.8.1 Rationale

The simulation web service is designed around the concept of **experiment**.

An experiment, identified by a unique identifier, is composed of two entities:

- a network, and
- one (or more) configured diffusion models.

Experiments are used to keep track of the simulation definition, to return consecutive model iterations to the user and to store - locally on the experiment server - the current status of the diffusion process.

In particular, in order to perform an experiment, a user must:

1. Request a token, which univocally identifies the experiment;
2. Select or load a network resource;
3. Select one, or more, diffusion model(s);
4. (optional) Use the advanced configuration facilities to define node/edge parameters;
5. Execute the step-by-step simulation;
6. (optional) Reset the experiment status, modify the models/network;
7. Destroy the experiment.

The last action, involving the destruction of the experiment, is designed to clean the serialization made by the service of the incremental experiment status.

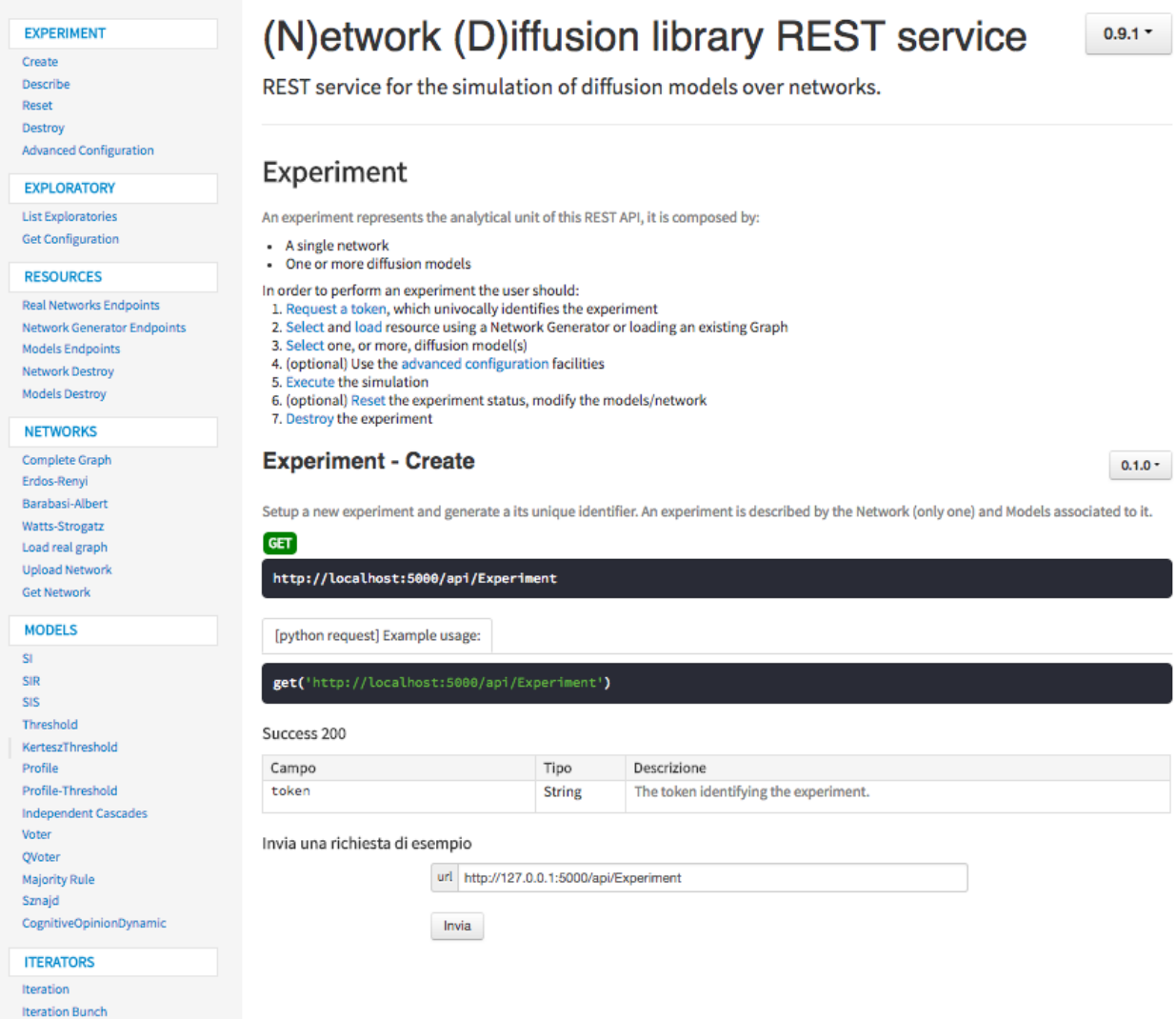

Fig. 9: REST interactive documentation page.

If an experiment is not explicitly destroyed its data is removed, and the associated token invalidated, after a temporal window that can be configured by the service administrator.

`NDlib-REST` is shipped as a Docker container image so to make it configuration free and easier to setup. Moreover, the simulation server is, by default, executed within a Gunicorn instance allowing parallel executions of multiple experiments at the same time.

`NDlib-REST` is built using Flask and offers a standard online documentation page that can also be directly used to test the exposed endpoints both configuring and running experiments.

API Interface

As a standard for REST services, all the calls made to `NDlib-REST` endpoints generate JSON responses.

The APIs of the simulation service are organized in six categories so to provide a logic separation among all the exposed resources. In particular, in `NDlib-REST` are exposed endpoints handling:

- **Experiment**: endpoints in this category allow to create, destroy, configure, reset and describe experiments;
- **Exploratories**: endpoints in this category allow to load predefined scenarios (e.g. specific networks/models with explicit initial configuration);
- **Resources**: endpoints in this category allow to query the system to dynamically discover the endpoints (and their descriptions) defined within the system;
- **Networks**: endpoints in this category handle a load of network data as well as the generation of synthetic graphs (Barabasi-Albert, Erdos-Renyi, Watts-Strogatz. . .);
- **Models**: endpoints in this category expose the `NDlib` models;
- **Iterators**: endpoints in this category expose the step-by-step and iteration bunch facilities needed to run the simulation.

The simulation service allows to attach multiple diffusion models to the same experiment, thus both the single iteration and the iteration bunch endpoints expose additional parameters that allow the user to select the models for which the call was invoked.

By default, when such parameter is not specified, all the models are executed and their incremental statuses returned.

A particular class of endpoints is the **Exploratories** one. Such endpoints are used to define the access to pre-set diffusion scenarios. Using such facilities the owner of the simulation server can describe, beforehand, specific scenarios, package them and make them available to the service users.

From an educational point of view such mechanism can be used, for instance, by professors to design emblematic diffusion scenarios (composed by both network and initial node/edge statuses) so to let the students explore their impact on specific models configurations (e.g. to analyze the role of weak-ties and/or community structures).

1.8.2 Installation

The project provides:

- **The REST service: ndrest.py**
  - Web API docs: http://127.0.0.1:5000/docs
  - Unittest: ndlib-rest/service_test
- Python REST client: ndlib-rest/client

REST service setup

Local testing

```
python ndrest.py
```

Local testig with multiple workers (using gunicorn web server):

```
gunicorn -w num_workers -b 127.0.0.1:5000 ndrest:app
```

In order to change the binding IP/port modify the apidoc.json file. To update the API page run the command:

```
apidoc -i ndlib-rest/ -o ndlib-rest/static/docs
```

Docker Container

The web application is shipped in a Docker container. You can use the Dockerfile to create a new image and run the web application using the gunicorn application server.

To create the Docker image, install Docker on your machine. To create the image execute the following command from the local copy of the repository

```
docker build -t [tagname_for_your_image] .
```

The command create a new image with the specified name. Pay attention to the **.** a the end of the command.

```
docker run -d -i -p 5000:5000 [tagname_for_your_image]
```

This command execute a container with the previous image, bind the local port 5000 to the internal port of the container. The option **-d** make the container to run in the background (detached)

To have a list of all active container

```
docker ps -al
```

To stop a container

```
docker stop container_name
```

1.8.3 Configuration

In ndrest.py are specified limits for graph sizes.

In particular are set the minimum and maximum numbers of nodes (for both generators and loaded networks) as well as the maximum file sizes for upload.

```
app.config['MAX_CONTENT_LENGTH'] = 50 * 1024 * 1024  # 50MB limit for uploads
max_number_of_nodes = 100000
min_number_of_nodes = 200 # inherited by networkx
```

- The "complete graph generator" endpoint represents the only exception to the specified lower bound on number of nodes: such model lowers the minimum to 100 nodes. Indeed, the suggested limits can be increased to handle bigger graphs.
- When loading external graphs nodes MUST be identified by integer ids.

1.9 Visual Framework

NDlib aims to an heterogeneous audience composed by technicians as well as analysts. In order to abstract from the its programming interface we built a simple visual framework that allows to simulate NDlib built-in models on synthetic graphs.

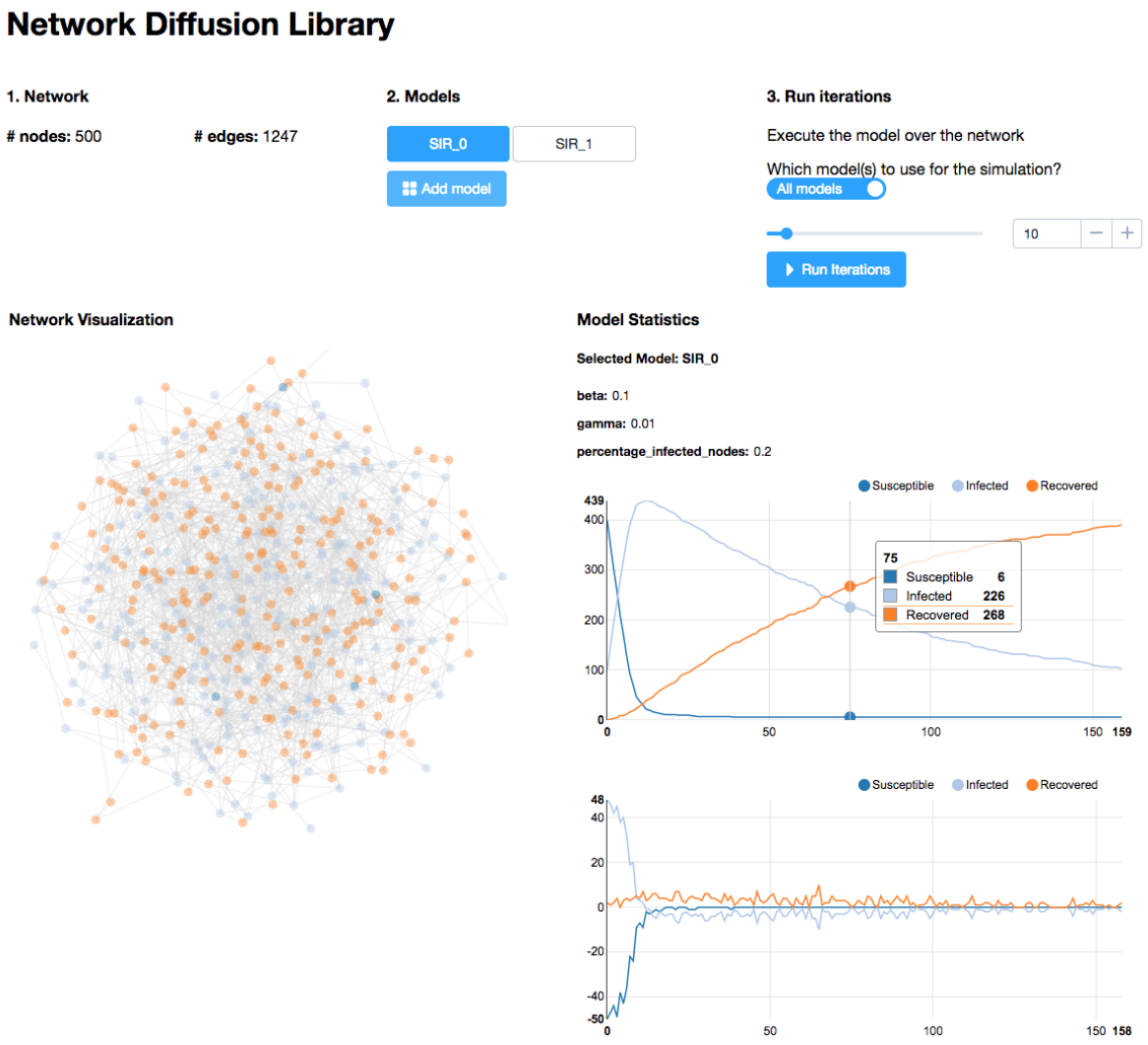

Fig. 10: Visual Framework.

Project Website: https://github.com/GiulioRossetti/NDLib_viz

1.9.1 Rationale

NDlib-Viz aims to make non-technicians able to design, configure and run epidemic simulations, thus removing the barriers introduced by the usual requirements of programming language knowledge. Indeed, apart from the usual research-oriented audience, we developed NDlib-Viz to support students and facilitate teachers to introduce

epidemic models. The platform itself is a web application: it can be executed on a local as well as on a remote `NDlib-REST` installation.

1.9.2 Installation

`NDlib-Viz` requires a local active instance of `NDlib-REST` to be executed.

```
# install dependencies
npm install

# serve with hot reload at localhost:8080
npm run dev

# build for production with minification
npm run build

# build for production and view the bundle analyzer report
npm run build --report
```

For detailed explanation on how things work, checkout the guide and docs for vue-loader.

1.9.3 Architecture

The Visualization Framework is a single-page web application implemented using Javascript and HTML 5. The decoupling of the simulation engine and the visual interface allows us to exploit modern browsers to provide an efficient environment for visualization of models and interactions.

- The structure and layout of the page are managed with Bootstrap.
- The business logic and visualization of graphical widgets are implemented in D3.js.
- Nodes and edges of the networks are drawn using the Force Layout library provided by the D3 library.
- The network visualization is implemented using Canvas object provided by standard HTML5. This allows a very efficient update of the network view.
- The charts showing the Diffusion Trend and Prevalence are created using NVD3 library.

The Visualization Framework is implemented using a Model-Control-View (MCV) design pattern. The model is managed by a central component that implements a REST API client that handle the status of the experiment. When the user interacts with one of the views (charts, network layout, toolbar), the controller notifies the model to update the experiment. Each interaction with the visual interface is managed by the model component that centralizes all the communications with the REST server. The calls to the server are executed asynchronously, and the component updates the visual interface as soon as a response arrives from the server.

1.10 Developer Guide

1.10.1 Working with NDlib source code

Contents:

Introduction

These pages describe a git and github workflow for the NDlib project.

There are several different workflows here, for different ways of working with NDlib.

This is not a comprehensive git reference, it's just a workflow for our own project. It's tailored to the github hosting service. You may well find better or quicker ways of getting stuff done with git, but these should get you started.

For general resources for learning git, see git resources.

Install git

Overview

Debian / Ubuntu	`sudo apt-get install git`
Fedora	`sudo yum install git-core`
Windows	Download and install msysGit[1]
OS X	Use the git-osx-installer[2]

In detail

See the git page for the most recent information.

Have a look at the github install help pages available from *github help*[3] .

There are good instructions here: http://book.git-scm.com/2_installing_git.html

Following the latest source

These are the instructions if you just want to follow the latest NDlib source, but you don't need to do any development for now.

The steps are:

- *Install git*
- get local copy of the ndlib github git repository
- update local copy from time to time

Get the local copy of the code

From the command line:

```
git clone git://github.com/GiulioRossetti/ndlib.git
```

You now have a copy of the code tree in the new `ndlib` directory.

[1] https://git-for-windows.github.io
[2] https://code.google.com/archive/p/git-osx-installer/downloads
[3] http://help.github.com/

Updating the code

From time to time you may want to pull down the latest code. It is necessary to add the NDlib repository as a remote to your configuration file. We call it upstream.

git remote set-url upstream https://github.com/GiulioRossetti/ndlib.git

Now git knows where to fetch updates from.

cd ndlib git fetch upstream

The tree in `ndlib` will now have the latest changes from the initial repository, unless you have made local changes in the meantime. In this case, you have to merge.

git merge upstream/master

It is also possible to update your local fork directly from GitHub:

1. Open your fork on GitHub.
2. Click on 'Pull Requests'.
3. Click on 'New Pull Request'. By default, GitHub will compare the original with your fork. If you didn't make any changes, there is nothing to compare.
4. Click on 'Switching the base' or click 'Edit' and switch the base manually. Now GitHub will compare your fork with the original, and you should see all the latest changes.
5. Click on 'Click to create a pull request for this comparison' and name your pull request.
6. Click on Send pull request.
7. Scroll down and click 'Merge pull request' and finally 'Confirm merge'. You will be able to merge it automatically unless you did not change you local repo.

Making a patch

You've discovered a bug or something else you want to change in ndlib .. - excellent!

You've worked out a way to fix it - even better!

You want to tell us about it - best of all!

The easiest way is to make a *patch* or set of patches. Here we explain how.

Making patches

Overview

```
# tell git who you are
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
# get the repository if you don't have it
git clone git://github.com/GiulioRossetti/ndlib.git
# make a branch for your patching
cd networkx
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
# hack, hack, hack
```

(continues on next page)

(continued from previous page)

```
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
# make the patch files
git format-patch -M -C master
```

Then, open an issue on the projetc GitHub and attach the generated patch files.

In detail

1. Tell git who you are so it can label the commits you've made:

```
git config --global user.email you@yourdomain.example.com
git config --global user.name "Your Name Comes Here"
```

2. If you don't already have one, clone a copy of the ndlib repository:

```
git clone git://github.com/GiulioRossetti/ndlib.git
cd networkx
```

3. Make a 'feature branch'. This will be where you work on your bug fix. It's nice and safe and leaves you with access to an unmodified copy of the code in the main branch:

```
git branch the-fix-im-thinking-of
git checkout the-fix-im-thinking-of
```

4. Do some edits, and commit them as you go:

```
# hack, hack, hack
# Tell git about any new files you've made
git add somewhere/tests/test_my_bug.py
# commit work in progress as you go
git commit -am 'BF - added tests for Funny bug'
# hack hack, hack
git commit -am 'BF - added fix for Funny bug'
```

   Note the `-am` options to `commit`. The `m` flag just signals that you're going to type a message on the command line.

5. When you have finished, check you have committed all your changes:

```
git status
```

6. Finally, make your commits into patches. You want all the commits since you branched from the `master` branch:

```
git format-patch -M -C master
```

   You will now have several files named for the commits:

```
0001-BF-added-tests-for-Funny-bug.patch
0002-BF-added-fix-for-Funny-bug.patch
```

Attach these files to a novel issue on the project GitHub.

When you are done, to switch back to the main copy of the code, just return to the `master` branch:

```
git checkout master
```

1.10.2 Extend NDlib

The `NDlib` library can be extended by adding both models and visualization facilities.

In this section are introduced the basilar concept behind the class model adopted in `NDlib` and some best practice for the definition of novel models and visualizations.

Describe a Diffusion Model

All the diffusion models implemented in `NDlib` extends the abstract class `ndlib.models.DiffusionModel`.

class `ndlib.models.DiffusionModel.`**DiffusionModel**(*graph*, *seed=None*)
Partial Abstract Class that defines Diffusion Models

Such class implements the logic behind model construction, configuration and execution.

In order to describe a novel diffusion algorithm the following steps must be followed:

Model Description

As convention a new model should be described in a python file named after it, e.g. a `MyModule` class should be implemented in a `MyModule.py` file.

`DiffusionModel.`**__init__**(*self*, *graph*)
Model Constructor

Parameters **graph** – A networkx graph object

In oder to effectively describe the model the `__init__` function of `ndlib.models.DiffusionModel` must be specified as follows:

```
import future.utils
import numpy as np
import networkx as nx
from ndlib.models.DiffusionModel import DiffusionModel

class MyModel(DiffusionModel):

        def __init__(self, graph):

                # Call the super class constructor
                super(self.__class__, self).__init__(graph)

                # Method name
                self.name = "MyModel"

                # Available node statuses
                self.available_statuses = {
                        "Status_0": 0,
                        "Status_1": 1
```

(continues on next page)

(continued from previous page)

```
                }
                # Exposed Parameters
                self.parameters = {
                        "model":
                                "parameter_name": {
                                        "descr": "Description 1",
                                        "range": [0,1],
                                        "optional": False
                                        },
                                },
                        "nodes":
                                "node_parameter_name": {
                                        "descr": "Description 2",
                                        "range": [0,1],
                                        "optional": True
                                        },
                                },
                        "edges":
                                "edge_parameter_name": {
                                                "descr": "Description 3",
                                                "range": [0,1],
                                                "optional": False
                                        },
                                },
                        }
```

In the `__init__` methods three components are used to completely specify the model:

- `self.name`: its **name**;
- `self.available_statuses`: the node **statuses** it allows along with an associated numerical code;
- `self.parameters`: the **parameters** it requires, their range, description and optionality.

All those information will be used to check the user provided configurations as well as metadata for visualizations.

Iteration Rule

Once described the model metadata it is necessary to provide the agent-based description of its general iteration-step.

`DiffusionModel.`**`iteration`**(*self*)
Execute a single model iteration

Parameters **`node_status`** – if the incremental node status has to be returned.

Returns Iteration_id, (optional) Incremental node status (dictionary node->status), Status count (dictionary status->node count), Status delta (dictionary status->node delta)

To do so, the `iteration()` method of the base class has to be overridden in `MyModel` as follows:

```
def iteration(self, node_status=True):

        self.clean_initial_status(self.available_statuses.values())
        actual_status = {node: nstatus for node, nstatus in self.status.iteritems()}

        # if first iteration return the initial node status
        if self.actual_iteration == 0:
                self.actual_iteration += 1
```

(continues on next page)

(continued from previous page)

```
                delta, node_count, status_delta = self.status_delta(actual_status)
                if node_status:
                        return {"iteration": 0, "status": actual_status.copy(),
                                        "node_count": node_count.copy(), "status_delta
↪": status_delta.copy()}
                else:
                        return {"iteration": 0, "status": {},
                                        "node_count": node_count.copy(), "status_delta
↪": status_delta.copy()}


        # iteration inner loop
        for u in self.graph.nodes():
                # evluate possible status changes using the model parameters␣
↪(accessible via self.params)
                # e.g. self.params['beta'], self.param['nodes']['threshold'][u], self.
↪params['edges'][(id_node0, idnode1)]

        # identify the changes w.r.t. previous iteration
        delta, node_count, status_delta = self.status_delta(actual_status)

        # update the actual status and iterative step
        self.status = actual_status
        self.actual_iteration += 1

        # return the actual configuration (only nodes with status updates)
        if node_status:
                return {"iteration": self.actual_iteration - 1, "status": delta.
↪copy(),
                                "node_count": node_count.copy(), "status_delta":␣
↪status_delta.copy()}
        else:
                return {"iteration": self.actual_iteration - 1, "status": {},
                "node_count": node_count.copy(), "status_delta": status_delta.copy()}
```

The provided template is composed by 4 steps:

1. first iteration handling: if present the model returns as result of the first iteration is initial status;
2. making a copy of the actual diffusion status;
3. iteration loop: definition, and application, of the rules that regulates individual node status transitions;
4. construction of the incremental result.

All the steps are mandatory in order to assure a consistent behaviour across different models

All the user specified parameters (models as well as nodes and edges ones) can be used within the `iteration()` method: to access them an internal data structure is provided, `self.params`.

`self.params` is a dictionary that collects all the passed values using the following notation:

- Model parameters: `self.params['model']['parameter_name']`
- Node parameters: `self.param['nodes']['nodes_parameter'][node_id]`
- Edge parameters: `self.param['edges']['edges_parameter'][(node_id1, node_id2)]`

Within the iteration loop the node status updates must be made on the `actual_status` data structure, e.g. the copy made during Step 1.

Each iteration returns the **incremental** status of the diffusion process as well as the iteration **progressive number**.

Describe a visualization

All the `matplotlib` visualizations implemented so far in `NDlib` extends the abstract class `nndlib.viz.mpl.DiffusionViz.DiffusionPlot`.

class `ndlib.viz.mpl.DiffusionViz.`**DiffusionPlot**(*model*, *trends*)

Conversely, visualizations that use the `bokeh` library, should extend the abstract class `nndlib.viz.bokeh.DiffusionViz.DiffusionPlot`.

class `ndlib.viz.bokeh.DiffusionViz.`**DiffusionPlot**(*model*, *trends*)

Here is introduced the pattern for describing novel `matplotlib` based visualization, `bokeh` ones following the same rationale.

So far `DiffusionPlot` implements the visualization logic *only* for generic **trend line plot** built upon simulation iterations and model metadata.

Line Plot Definition

As convention a new visualization should be described in a python file named after it, e.g. a `MyViz` class should be implemented in a `MyViz.py` file.

`DiffusionPlot.`**__init__**(*self*, *model*, *iteration*)
Initialize self. See help(type(self)) for accurate signature.

In oder to effectively describe the visualization the `__init__` function of `ndlib.viz.bokeh.DiffusionViz.DiffusionPlot` must be specified as follows:

```
from ndlib.viz.mpl.DiffusionViz import DiffusionPlot

class MyViz(DiffusionPlot):

        def __init__(self, model, trends):
                super(self.__class__, self).__init__(model, trends)
                self.ylabel = "#Nodes"
                self.title = "Diffusion Trend"
```

Data Preparation

Once described the plot metadata it is necessary to prepare the data to be visualized through the `plot()` method.

To do so, the `iteration_series(percentile)` method of the base class has to be overridden in `MyViz`.

`DiffusionPlot.`**iteration_series**(*self*, *percentile*)
Prepare the data to be visualized

Parameters **percentile** – The percentile for the trend variance area

Returns a dictionary where iteration ids are keys and the associated values are the computed measures

Such method can access the trend data, as returned by `ndlib.models.DiffusionModel.DiffusionModel.build_trends(self, iterations)` in `self.iterations`.

1.11 Bibliography

NDlib was developed for research purposes.

So far it has been used/cited by the following publications:

"NDlib: a Python Library to Model and Analyze Diffusion Processes Over Complex Networks" G. Rossetti, L. Milli, S. Rinzivillo, A. Sirbu, D. Pedreschi, F. Giannotti. International Journal of Data Science and Analytics. 2017. DOI:0.1007/s41060-017-0086-6 (pre-print available on arXiv)

"NDlib: Studying Network Diffusion Dynamics" G. Rossetti, L. Milli, S. Rinzivillo, A. Sirbu, D. Pedreschi, F. Giannotti. IEEE International Conference on Data Science and Advanced Analytics, DSAA. 2017.

"Information Diffusion in Complex Networks: The Active/Passive Conundrum" L. Milli, G. Rossetti, D. Pedreschi, F. Giannotti International Conference on Complex Networks and their Applications, 2017. DOI:10.1007/978-3-319-72150-7_25

"Active and passive diffusion processes in complex networks." Milli, L., Rossetti, G., Pedreschi, D., & Giannotti, F. Applied network science, 3(1), 42, 2018.

"Diffusive Phenomena in Dynamic Networks: a data-driven study" L. Milli, G. Rossetti, D. Pedreschi, F. Giannotti. 9th Conference on Complex Networks, CompleNet, 2018.

"Stochastic dynamic programming heuristics for influence maximization–revenue optimization." Lawrence, Trisha, and Patrick Hosein. International Journal of Data Science and Analytics (2018): 1-14.

"Optimization of the Choice of Individuals to Be Immunized Through the Genetic Algorithm in the SIR Model" Rodrigues, R. F., da Silva, A. R., da Fonseca Vieira, V., AND Xavier, C. R. In International Conference on Computational Science and Its Applications (pp. 62-75), 2018.

"Algorithmic bias amplifies opinion fragmentation and polarization: A bounded confidence model" Sîrbu, A., Pedreschi, D., Giannotti, F., & Kertész, J. PloS one, 14(3), 2019.

"Similarity forces and recurrent components in human face-to-face interaction networks." Flores, Marco Antonio Rodríguez, and Fragkiskos Papadopoulos. Physical review letters 121.25, 2018.

"Resampling-based predictive simulation framework of stochastic diffusion model for identifying top-K influential nodes." Ohara, K., Saito, K., Kimura, M., & Motoda, H. International Journal of Data Science and Analytics, 1-21, 2019.

"Learning Data Mining." Guidotti, R., Monreale, A., & Rinzivillo, S. (2018, October). In IEEE 5th International Conference on Data Science and Advanced Analytics (DSAA) (pp. 361-370), 2018.

Index

Symbols

A

C

D

E

G

K

M

O

P

Q

R

S

T

V