Network Diffusion

Release 0.13.0

Michał Czuba, Piotr Bródka

Sep 30, 2023

CONTENTS

Network Diffusion is a library that allows to design and run diffusion phenomena processes in networks. The package has been built based on networkx and is fully compatible. With Network Diffusion, the user can work with multi- and single-layer networks, define propagation models from scratch, use predefined ones, and perform simulations.

Please cite this library as:

```
@INPROCEEDINGS{czuba2022networkdiffusion,
     author={Czuba, Micha\l{} and Br\'{o}dka, Piotr},
     booktitle={
        2022 IEEE 9th International Conference on Data Science and
        Advanced Analytics (DSAA)
     },
     title={
        Simulating Spreading of Multiple Interacting Processes in
        Complex Networks
     },
     year={2022},
     month={oct},
     volume={},
     number={},
     pages={1-10},
     publisher={IEEE},
     address={Shenzhen, China},
     doi={10.1109/DSAA54385.2022.10032425},
}
```

Feel free to contribute! We strongly believe in open-source projects. Hence our library provides open interfaces for new models, metrics, and functions. If you need to implement a piece of code and, by that, enhance the package, please let us know in the form of a pull request. In case of any questions, do not hesitate to contact us: **michal.czuba@pwr.edu.pl**

CHAPTER

ONE

CONTENTS OF THE WEBSITE

1.1 Quick info

1.1.1 Information about this project

This project has been created due to the lack of Python tools, which allow performing process-propagation experiments in the networks (graphs). The current version of the library contains a multi-process spreading toolset for discrete phenomena (like 'SIS').

1.1.2 Github repository

All code can be found on GitHub repo.

1.1.3 Code Ocean capsule

There is an option to make a dry run of the package using an interactive capsule published at CodeOcean. Visit this page to play with Network Diffusion!

1.1.4 Experiments on LTM model

Here is another project which bases on this package. You can find there pipelines that evaluate effectivness of various seed selection methods (like Page Rank or Degree Centrality) on Linear Threshold Model.

1.2 Installation

Package is available to install via PyPi and Conda

1.2.1 pip

To install via PyPi run:

pip install network_diffusion

1.2.2 conda

To instal via conda run:

conda install -c anty-filidor network_diffusion

1.3 Reference guide

1.3.1 Operations on multilayer networks

See dedicated multilayer guide for these functions.

Class `MLNetworkActor`

Implemented in `network_diffusion.mln.actor`.

class MLNetworkActor(*actor_id: str*, *layers_states: Dict[str, str]*)

Dataclass that contain data of actor in the network.

property layers: Tuple[str, ...]

Get network layers where actor exists.

property states: Dict[str, str]

Get actor's states for where actitor exists.

states_as_compartmental_graph() → Tuple[str, ...]

Return actor states in form accepted by CompartmentalGraph.

Returns

a tuple in form on ('process_name.state_name', …), e.g. ('awareness.UA', 'illness.I', 'vaccination.V')

Class `MultilayerNetwork`

Implemented in `network_diffusion.mln.mlnetwork`.

class MultilayerNetwork(*layers: Dict[str, Graph]*)

Container for multilayer network.

copy() → *MultilayerNetwork*

Create a deep copy of the network.

classmethod from_mpx(*file_path: str*) → *MultilayerNetwork*

Load multilayer network from mpx file.

Note, that is omits some non-important attributes of network defined in the file, i.e. node attributes.

Parameters
file_path – path to the file

classmethod from_nx_layer(*network_layer: Graph*, *layer_names: List[Any]*) → *MultilayerNetwork*

Create multiplex network from one nx.Graph layer and layers names.

Note that *network_layer* is replicated through all layers.

Parameters

- **network_layer** – basic layer which is replicated through all ones
- **layer_names** – names for layers in multiplex network

classmethod from_nx_layers(*network_list: List[Graph]*, *layer_names: List[Any] | None = None*) → *MultilayerNetwork*

Load multilayer network as list of layers and list of its labels.

Parameters

- **network_list** – list of nx networks
- **layer_names** – list of layer names. It can be none, then labels are set automatically

get_actor(*actor_id: Any*) → *MLNetworkActor*

Get actor data basing on its name.

get_actors(*shuffle: bool = False*) → List[*MLNetworkActor*]

Get actors that exist in the network and read their states.

Parameters
shuffle – a flag that determines whether to shuffle actor list

Returns
a list with actors that live in the network

get_actors_num() → int

Get number of actors that live in the network.

get_layer_names() → List[str]

Get names of layers in the network.

Returns
list of layers' names

get_links(*actor_id: Any | None = None*) → Set[Tuple[*MLNetworkActor*, *MLNetworkActor*]]

Get links connecting all actors from the network regardless layers.

Returns
a set with edges between actors

get_nodes_num() → Dict[str, int]

Get number of nodes that live in each layer of the network.

is_directed() → bool

Check whether at least one layer is a DirectedGraph.

is_multiplex() → bool

Check if network is multiplex.

subgraph(*actors: List[MLNetworkActor]*) → *MultilayerNetwork*

Return a subgraph of the network.

The induced subgraph of the graph contains the nodes in *nodes* and the edges between those nodes. This is an equivalent of nx.Graph.subgraph.

to_multiplex() → *MultilayerNetwork*

Convert network to multiplex one by adding missing nodes.

Auxiliary functions for operations on `MultilayerNetwork`

Implemented in `network_diffusion.mln.functions`.

Script with functions of NetworkX extended to multilayer networks.

all_neighbors(*net:* MultilayerNetwork, *actor:* MLNetworkActor) → Iterator[*MLNetworkActor*]

Return all of the neighbors of an actor in the graph.

If the graph is directed returns predecessors as well as successors. Overloads networkx.classes.functions.all_neighbours.

betweenness(*net:* MultilayerNetwork) → Dict[*MLNetworkActor*, float]

Return value of mean betweennes centrality for actors layers.

closeness(*net:* MultilayerNetwork) → Dict[*MLNetworkActor*, float]

Return value of mean closeness centrality for actors layers.

core_number(*net:* MultilayerNetwork) → Dict[*MLNetworkActor*, int]

Return the core number for each actor.

A k-core is a maximal subgraph that contains actors of degree k or more. A core number of a node is the largest value k of a k-core containing that node. Not implemented for graphs with parallel edges or self loops. Overloads networkx.algorithms.core.core_number.

Parameters

net – multilayer network

Returns

dictionary keyed by actor to the core number.

degree(*net:* MultilayerNetwork) → Dict[*MLNetworkActor*, int]

Return number of connecting links per all actors from the network.

get_toy_network() → *MultilayerNetwork*

Get threelayered toy network easy to visualise.

k_shell_mln(*net:* MultilayerNetwork, *k: int | None = None*, *core_number: Dict[MLNetworkActor, int] | None = None*) → *MultilayerNetwork*

Return the k-shell of net with degree computed actorwise.

The k-shell is the subgraph induced by actors with core number k. That is, actors in the k-core that are not in the (k+1)-core. The k-shell is not implemented for graphs with self loops or parallel edges. Overloads networkx.algorithms.core.k_shell.

Parameters

- **net** – A graph or directed graph.
- **k** – The order of the shell. If not specified return the outer shell.

- **core_number** – Precomputed core numbers keyed by node for the graph *net*. If not specified, the core numbers will be computed from *net*.

Returns
The k-shell subgraph

katz(*net: MultilayerNetwork*) → Dict[*MLNetworkActor*, float]

Return value of mean Katz centrality for actors layers.

multiplexing_coefficient(*net: MultilayerNetwork*) → float

Compute multiplexing coefficient.

Multiplexing coefficient is defined as proportion of number of nodes common to all layers to number of all unique nodes in entire network

Returns
(float) multiplexing coefficient

neighbourhood_size(*net: MultilayerNetwork*, *connection_hop: int = 1*) → Dict[*MLNetworkActor*, int]

Return n-hop neighbourhood sizes of all actors from the network.

number_of_selfloops(*net: MultilayerNetwork*) → int

Return the number of selfloop edges in the entire network.

A selfloop edge has the same node at both ends. Overloads networkx.classes. functions.number_of_selfloops.

squeeze_by_neighbourhood(*net: MultilayerNetwork*) → Graph

Squeeze multilayer network to single layer by neighbourhood of actors.

All actors are preserved, links are produced according to naighbourhood between actors regardless layers.

Parameters
net – a multilayer network to be squeezed

Returns
a networkx.Graph representing *net*

voterank_actorwise(*net: MultilayerNetwork*, *number_of_actors: int | None = None*) → List[*MLNetworkActor*]

Select a list of influential ACTORS in a graph using VoteRank algorithm.

VoteRank computes a ranking of the actors in a graph based on a voting scheme. With VoteRank, all actors vote for each of its neighbours and the actor with the highest votes is elected iteratively. The voting ability of neighbors of elected actors is decreased in subsequent turns. Overloads networkx.algorithms.core.k_shell.

Parameters

- **net** – multilayer network
- **number_of_actors** – number of ranked actors to extract (default all).

Returns
ordered list of computed seeds, only actors with positive number of votes are returned.

1.3.2 Operations on temporal networks

In the library TemporalNetwork is an ordered sequence of MultilayerNetworks. In base scenario one can obtain classic temporal network by having a chain of one-layered MultilayerNetworks.

Class `TemporalNetwork`

Implemented in `network_diffusion.tpn.tpnetwork`.

class TemporalNetwork(*snaps: List[MultilayerNetwork]*)

Container for a temporal network.

classmethod from_cogsnet(*forgetting_type: str*, *snapshot_interval: int*, *edge_lifetime: int*, *mu: float*, *theta: float*, *units: int*, *path_events: str*, *delimiter: str*) → *TemporalNetwork*

Load events from a csv file and create CogSNet.

Note, the csv file should be in the form of: SRC DST TIME. The timestamps TIME should be in ascending order. The timestamps are expected to be provided in seconds.

Parameters

- **forgetting_type** (*str*) – The forgetting function used to decrease the weight of edges over time. Allowed values are 'exponential', 'power', or 'linear'.
- **snapshot_interval** (*int*) – The interval for taking snapshots (0 or larger) expressed in units param (seconds, minutes, hours). A value of 0 means taking a snapshot after each event.
- **edge_lifetime** (*int*) – The lifetime of an edge after which the edge will disappear if no new event occurs (greater than 0).
- **mu** (*float*) – The value of increasing the weight of an edge for each new event (greater than 0 and less than or equal to 1).
- **theta** (*float*) – The cutoff point (between 0 and mu). If the weight falls below theta, the edge will disappear.
- **units** (*int*) – The time units (1 for seconds, 60 for minutes, or 3600 for hours). For the power forgetting function, this parameter also determines the minimum interval between events to prevent them from being skipped when calculating the weight.
- **path_events** (*str*) – The path to the CSV file with events.
- **delimiter** (*str*) – The delimiter for the CSV file (allowed values are ',', ';', or '\t').

classmethod from_nx_layers(*network_list: List[Graph]*, *snap_ids: List[Any] | None = None*) → *TemporalNetwork*

Load a temporal network from a list of networks and their snapshot ids.

Parameters

- **network_list** – a list of nx networks
- **snap_ids** – list of snapshot ids. It can be none, then ids are set automatically, if not, then snapshots will be sorted according to snap_ids list

classmethod from_txt(*file_path: str*, *time_window: int*, *directed: bool = True*) → *TemporalNetwork*

Load a temporal network from a txt file.

Note, the txt file should be in the form of: SRC DST UNIXTS. The timestamps UNIXTS should be in ascending order. The timestamps are expected to be provided in seconds.

Parameters

- `file_path` – path to the file
- `time_window` – the time window size for each snapshot
- `directed` – indicate if the graph is directed

`get_actors`(*shuffle: bool = False*) → List[*MLNetworkActor*]

Get actors that from the first snapshot of network.

Parameters
`shuffle` – a flag that determines whether to shuffle actor list

`get_actors_from_snap`(*snapshot_id: int*, *shuffle: bool = False*) → List[*MLNetworkActor*]

Get actors that exist in the network at given snapshot.

Parameters

- `snapshot_id` – snapshot for which to take actors, starts from 0
- `shuffle` – a flag that determines whether to shuffle actor list

`get_actors_num`() → int

Get number of actors that live in the network.

1.3.3 Propagation models

See dedicated propagation model guide for these functions.

Base structures for concrete models

`class BaseModel`(*compartmental_graph:* CompartmentalGraph, *seed_selector:* BaseSeedSelector)

Base abstract propagation model.

`abstract agent_evaluation_step`(*agent: Any*, *layer_name: str*, *net:* MultilayerNetwork) → str

Try to change state of given node of the network according to model.

Parameters

- `agent` – id of the node or the actor to evaluate
- `layer_name` – a layer where the node exists
- `net` – a network where the node exists

Returns
state of the model after evaluation

`property compartments:` *`CompartmentalGraph`*

Return defined compartments and allowed transitions.

`abstract determine_initial_states`(*net:* MultilayerNetwork) → List[NetworkUpdateBuffer]

Determine initial states in the network according to seed selector.

Parameters
`net` – network to initialise seeds for

Returns
list of nodes with their states

abstract get_allowed_states(*net: MultilayerNetwork*) → Dict[str, Tuple[str, ...]]

Return dict with allowed states in each layer of net if applied model.

Parameters
net – a network to determine allowed nodes' states for

static get_states_num(*net: MultilayerNetwork*) → Dict[str, Tuple[Tuple[Any, int], ...]]

Return states in the network with number of agents that adopted them.

It is the most basic function which assumes that field "status" in the network is self explaining and there is no need to decode it (e.g. to separate hidden state from public one).

Returns
dictionary with items representing each of layers and with summary of nodes states in values

abstract network_evaluation_step(*net: MultilayerNetwork*) → List[NetworkUpdateBuffer]

Evaluate the network at one time stamp according to the model.

Parameters
network – a network to evaluate

Returns
list of nodes that changed state after the evaluation

static update_network(*net: MultilayerNetwork, activated_nodes: List[NetworkUpdateBuffer]*) → List[Dict[str, str]]

Update the network global state by list of already activated nodes.

Parameters

- **net** – network to update
- **activated_nodes** – already activated nodes

class CompartmentalGraph

Class which encapsulates model of processes speared in network.

add(*process_name: str, states: List[str]*) → None

Add process with allowed states to the compartmental graph.

Parameters

- **layer** – name of process, e.g. "Illness"
- **type** – names of states like ['s', 'i', 'r']

compile(*background_weight: float = 0.0, track_changes: bool = False*) → None

Create transition matrices for models of propagation in each layer.

All transition probabilities are set to 0. To be more specific, transitions matrices are stored as a networkx one-directional graph. After compilation user is able to set certain transitions in model.

Parameters

- **background_weight** – [0,1] describes default weight of transition to make propagation more realistic by default it is set to 0
- **track_changes** – a flag to track progress of matrices creation

describe() → str

Print out parameters of the compartmental model.

Returns
returns string describing object,

`get_compartments()` → Dict[str, Tuple[str, ...]]

Get model parameters, i.e. names of layers and states in each layer.

Returns
dictionary keyed by names of layer, valued by tuples of states labels

`get_possible_transitions`(*state: Tuple[str, ...]*, *layer: str*) → Dict[str, float]

Return possible transitions from given state in given layer of model.

Note that possible transition is a transition with weight > 0.

Parameters

- **state** – state of the propagation model, i.e. ('awareness.UA', 'illness.I', 'vaccination.V')
- **layer** – name of the layer of propagation model from which possible transitions are being returned

Returns
dict with possible transitions in shape of: {possible different state in given layer: weight}

`get_seeding_budget_for_network`(*net:* MultilayerNetwork, *actorwise: bool = False*) → Dict[str, Dict[Any, int]]

Transform seeding budget from %s to numbers according to nodes/actors.

Parameters

- **net** – input network to convert seeding budget for
- **actorwise** – compute seeding budget for actors, else for nodes

Returns
dictionary in form as e.g.: {"ill": {"suspected": 45, "infected": 4, "recovered": 1}, "vacc": {"unvaccinated": 35, "vaccinated": 15}} for seeding_budget dict: {"ill": (90, 8, 2), "vacc": (70, 30)} and 50 nodes in each layer and nodewise mode.

property seeding_budget: Dict[str, Tuple[int | float | number, ...]]

Get seeding budget as % of the nodes in form of compartments as a dict.

E.g. something like that: {"ill": (90, 8, 2), "aware": (60, 40), "vacc": (70, 30)} for compartments such as: "ill": [s, i, r], "aware": [u, a], "vacc": [n, v]

`set_transition_canonical`(*layer: str*, *transition: EdgeView*, *weight: float*) → None

Set weight of certain transition in propagation model.

Parameters

- **layer** – name of the later in model
- **transition** – name of transition to be activated, edge in propagation model graph
- **weight** – in range (number [0, 1]) of activation

`set_transition_fast`(*initial_layer_attribute: str*, *final_layer_attribute: str*, *constraint_attributes: Tuple[str, ...]*, *weight: float*) → None

Set weight of certain transition in propagation model.

Parameters

- **initial_layer_attribute** – value of initial attribute which is being transited
- **final_layer_attribute** – value of final attribute which is being transition
- **constraint_attributes** – other attributes available in the propagation model

- **weight** – weight (in range [0, 1]) of activation

`set_transitions_in_random_edges`(*weights: List[List[float]]*) → None

Set out random transitions in propagation model using given weights.

Parameters
weights – list of weights to be set in random nodes e.g. for model of 3 layers that list [[0.1, 0.2], [0.03, 0.45], [0.55]] will change 2 weights in first layer, 2, i second and 1 in third

Concrete propagation models

Import from `network_diffusion.models`.

class DSAAModel(*compartmental_graph:* CompartmentalGraph)

Bases: *BaseModel*

This model implements algorithm presented at DSAA 2022.

`agent_evaluation_step`(*agent: Any*, *layer_name: str*, *net:* MultilayerNetwork) → str

Try to change state of given node of the network according to model.

Parameters

- **agent** – id of the node (here agent) to evaluate
- **layer_name** – a layer where the node exists
- **network** – a network where the node exists

Returns
state of the model after evaluation

`determine_initial_states`(*net:* MultilayerNetwork) → List[NetworkUpdateBuffer]

Set initial states in the network according to seed selection method.

Parameters
net – network to initialise seeds for

Returns
a list of state of the network after initialisation

`get_allowed_states`(*net:* MultilayerNetwork) → Dict[str, Tuple[str, ...]]

Return dict with allowed states in each layer of net if applied model.

In this model each process is binded with network's layer, hence we return just the compartments and allowed states.

Parameters
net – a network to determine allowed nodes' states for

`network_evaluation_step`(*net:* MultilayerNetwork) → List[NetworkUpdateBuffer]

Evaluate the network at one time stamp according to the model.

We ae updating nodes 'on the fly', hence the activated_nodes list is empty. This behaviour is due to intention to very reflect te algorithm presented at DSAA

Parameters
network – a network to evaluate

Returns
list of nodes that changed state after the evaluation

class MLTModel(*seeding_budget: Tuple[int | float | number, int | float | number]*, *seed_selector:* BaseSeedSelector, *protocol: str*, *mi_value: float*)

Bases: `BaseModel`

This model implements Multilayer Linear Threshold Model.

The model has been presented in paper: "Influence Spread in the Heterogeneous Multiplex Linear Threshold Model" by Yaofeng Desmond Zhong, Vaibhav Srivastava, and Naomi Ehrich Leonard. This implementation extends it to multilayer cases.

agent_evaluation_step(*agent:* MLNetworkActor, *layer_name: str*, *net:* MultilayerNetwork) → str

Try to change state of given actor of the network according to model.

Parameters

- **agent** – actor to evaluate in given layer
- **layer_name** – a layer where the actor exists
- **net** – a network where the actor exists

Returns

state of the actor in particular layer to be set after epoch

determine_initial_states(*net:* MultilayerNetwork) → List[NetworkUpdateBuffer]

Determine initial states in the net according to seed selection method.

Parameters

net – network to initialise seeds for

Returns

a list of nodes with their initial states

get_allowed_states(*net:* MultilayerNetwork) → Dict[str, Tuple[str, ...]]

Return dict with allowed states in each layer of net if applied model.

Parameters

net – a network to determine allowed nodes' states for

network_evaluation_step(*net:* MultilayerNetwork) → List[NetworkUpdateBuffer]

Evaluate the network at one time stamp with MLTModel.

Parameters

network – a network to evaluate

Returns

list of nodes that changed state after the evaluation

class MICModel(*seeding_budget: Tuple[int | float | number, int | float | number, int | float | number]*, *seed_selector:* BaseSeedSelector, *protocol: str*, *probability: float*)

Bases: `BaseModel`

This model implements Multilayer Independent Cascade Model.

agent_evaluation_step(*agent:* MLNetworkActor, *layer_name: str*, *net:* MultilayerNetwork) → str

Try to change state of given actor of the network according to model.

Parameters

- **agent** – actor to evaluate in given layer
- **layer_name** – a layer where the actor exists
- **net** – a network where the actor exists

Returns
state of the actor in particular layer to be set after epoch

determine_initial_states(*net: MultilayerNetwork*) → List[NetworkUpdateBuffer]

Set initial states in the network according to seed selection method.

Parameters
net – network to initialise seeds for

Returns
a list of state of the network after initialisation

get_allowed_states(*net: MultilayerNetwork*) → Dict[str, Tuple[str, ...]]

Return dict with allowed states in each layer of net if applied model.

Parameters
net – a network to determine allowed nodes' states for

network_evaluation_step(*net: MultilayerNetwork*) → List[NetworkUpdateBuffer]

Evaluate the network at one time stamp with MICModel.

Parameters
net – a network to evaluate

Returns
list of nodes that changed state after the evaluation

class TemporalNetworkEpistemologyModel(*seeding_budget: Tuple[int | float | number, int | float | number]*, *seed_selector: BaseSeedSelector*, *trials_nr: int*, *epsilon: float*)

Bases: `BaseModel`

Generalized version of Temporal Network Epistemology Model.

agent_evaluation_step(*agent: MLNetworkActor*, *layer_name: str*, *net: MultilayerNetwork*) → str

Try to change state of given actor of the network according to model.

Parameters

- **agent** – actor to evaluate in given layer
- **net** – a network where the actor exists
- **snapshot_id** – currently processed snapshot

Returns
state of the actor to be set in the next snapshot

static decode_actor_status(*encoded_status: str*) → Tuple[str, float, int]

Decode agent features from str form.

Parameters
encoded_status – a string representation of agent status

Returns
a tuple with agent state, belief level and evidence

determine_initial_states(*net: MultilayerNetwork*) → List[NetworkUpdateBuffer]

Set initial states in the network according to seed selection method.

Parameters
net – network to initialise seeds for

Returns
a list of state of the network after initialisation

`static encode_actor_status(`*state: str*, *belief: float*, *evidence: int*`)` → str

Encode agent features to str form.

Parameters

- **state** – state of an actor
- **belief** – level of agent's belief
- **evidence** – nr of successes drawn from binomial distribution in an experiment

Returns
a string representation of agent status

`get_allowed_states(`*net:* MultilayerNetwork`)` → Dict[str, Tuple[str, ...]]

Return dict with allowed states of net if applied model.

Parameters
net – a network to determine allowed nodes' states for

`static get_states_num(`*net:* MultilayerNetwork`)` → Dict[str, Tuple[Tuple[Any, int], ...]]

Return states in the network with number of agents that adopted them.

Vector of state for each agent is following:
<state of an actor><agent's belief><evidence>

And we are interested only in the state attribute.

Returns
dictionary with items representing each of layers and with summary of nodes states in values

`network_evaluation_step(`*net:* MultilayerNetwork`)` → List[NetworkUpdateBuffer]

Evaluate the given snapshot of the network.

Parameters
net – a network to evaluate

Returns
list of nodes that changed state after the evaluation

1.3.4 Performing experiments

See dedicated simulator guide for these functions.

Functions for logging experiments.

`class Logger(`*model_description: str*, *network_description: str*`)`

Store and processes logs acquired during performing Simulator.

`add_global_stat(`*log: Dict[str, Any]*`)` → None

Add raw log from single epoch to the object.

Parameters
log – raw log (i.e. a single call of MultiplexNetwork.get_states_num())

`add_local_stat(`*epoch: int*, *stats: List[Dict[str, str]]*`)` → None

Add local log from single epoch to the object.

convert_logs(*model_parameters: Dict[str, Tuple[str, ...]]*) → None

Convert raw logs into pandas dataframe.

Used after finishing aggregation of logs. It fulfills self._stats.

Parameters

model_parameters – parameters of the propagation model to store

plot(*to_file: bool = False*, *path: str | None = None*) → None

Plot out visualisation of performed experiment.

Parameters

- **to_file** – flag, if true save figure to file, otherwise it is plotted on screen
- **path** – path to save figure

report(*visualisation: bool = False*, *path: str | None = None*) → None

Create report of experiment.

It consists of report of the network, report of the model, record of propagation progress and optionally visualisation of the progress.

Parameters

- **visualisation** – (bool) a flag, if true visualisation is being plotted
- **path** – (str) path to folder where report will be saved if not provided logs are printed out on the screen

Functions for the phenomena spreading definition.

class Simulator(*model:* BaseModel, *network:* MultilayerNetwork | TemporalNetwork)

Perform experiment defined by PropagationModel on MultiLayerNetwork.

perform_propagation(*n_epochs: int*, *patience: int | None = None*) → *Logger*

Perform experiment on given network and given model.

It saves logs in Logger object which can be used for further analysis.

Parameters

- **n_epochs** – number of epochs to do experiment
- **patience** – if provided experiment will be stopped when in "patience" (e.g. 4) consecutive epoch there was no propagation

Returns

logs of experiment stored in special object

1.3.5 Seed selection classes for propagation models

See dedicated propagation model guide for these functions.

Base structures for concrete seed selectors

class BaseSeedSelector(***kwargs: Any*)

Bases: ABC

Base abstract class for seed selectors.

abstract static _calculate_ranking_list(*graph: Graph*) → List[Any]

Create a ranking of nodes based on concrete metric/heuristic.

Parameters
graph – single layer graph to compute ranking for

Returns
list of node-ids ordered descending by their ranking position

abstract actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Create actorwise ranking.

nodewise(*net:* MultilayerNetwork) → Dict[str, List[Any]]

Create nodewise ranking.

Concrete seed selectors

A definition of the seed selector based on degree centrality.

class DegreeCentralitySelector(***kwargs: Any*)

Bases: *BaseSeedSelector*

Degree Centrality seed selector.

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Get ranking for actors using Degree Centrality metric.

A definition of the seed selectors based on k-shell algorithm.

class KShellMLNSeedSelector(***kwargs: Any*)

Bases: *BaseSeedSelector*

Selector for MLTModel based on k-shell algorithm.

In contrary to KShellSeedSelector it utilises k-shell decomposition defined as in network_diffusion.mln.functions.k_shell_mln()

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Compute ranking for actors.

class KShellSeedSelector(***kwargs: Any*)

Bases: *BaseSeedSelector*

Selector for MLTModel based on k-shell algorithm.

According to "Seed selection for information cascade in multilayer networks" by Fredrik Erlandsson, Piotr Bródka, and Anton Borg we have extended k-shell ranking by combining it with degree of the node in each layer, so that ranking is better ordered (nodes in shells can be ordered).

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Compute ranking for actors.

A definition of "selector" that returns aprioiry provided actors.

class MockyActorSelector(*preselected_actors: List[MLNetworkActor]*)

Bases: `BaseSeedSelector`

Mocky seed selector - returns a ranking provided as argument to init.

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Get ranking for actors.

A definition of the seed selector based on neighbourhood size.

class NeighbourhoodSizeSelector(*connection_hop: int = 1*)

Bases: `BaseSeedSelector`

Neighbourhood Size seed selector.

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Get ranking for actors using Neighbourhood Size metric.

A definition of the seed selectors based on Page Rank algorithm.

class PageRankMLNSeedSelector(***kwargs: Any*)

Bases: `PageRankSeedSelector`

Selector for MLTModel based on Page Rank algorithm.

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Compute ranking for actors.

class PageRankSeedSelector(***kwargs: Any*)

Bases: `BaseSeedSelector`

Selector for MLTModel based on Page Rank algorithm.

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Compute ranking for actors.

Randomised seed selector.

class RandomSeedSelector(***kwargs: Any*)

Bases: `BaseSeedSelector`

Randomised seed selector prepared mainly for DSAA algorithm.

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Get actors randomly.

A definition of the seed selectors based on Vote Rank algorithm.

class VoteRankMLNSeedSelector(***kwargs: Any*)

Bases: `BaseSeedSelector`

Selector for MLTModel based on Vote Rank algorithm.

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Compute ranking for actors.

class VoteRankSeedSelector(***kwargs: Any*)

Bases: `BaseSeedSelector`

Selector for MLTModel based on Vote Rank algorithm.

actorwise(*net:* MultilayerNetwork) → List[*MLNetworkActor*]

Compute ranking for actors.

1.3.6 Auxiliary methods

Functions for the auxiliary operations.

create_directory(*dest_path: str*) → None

Check out if given directory exists and if doesn't it creates it.

Parameters
`dest_path` – absolute path to create folder

get_absolute_path() → str

Get absolute path of library.

get_nx_snapshot(*graph: DynGraph | DynDiGraph*, *snap_id: int*, *min_timestamp: int*, *time_window: int*) → Graph | DiGraph

Get an nxGraph typed snapshot for the given snapshot id.

Parameters

- `graph` – the dynamic graph
- `snap_id` – the snapshot id
- `min_timestamp` – the minimum timestamp in the graph
- `time_window` – the size of the time window

Returns
a snapshot graph of the given id

read_mpx(*file_path: str*) → Dict[str, List[Any]]

Handle MPX file for the MultilayerNetwork class.

Parameters
`file_path` – path to file

Returns
a dictionary with network to create class

read_tpn(*file_path: str*, *time_window: int*, *directed: bool = True*) → Dict[int, Graph | DiGraph]

Read temporal network from a text file for the TemporalNetwork class.

Parameters
`file_path` – path to file

Returns
a dictionary keyed by snapshot ID and valued by NetworkX Graph

1.4 Code usage examples

1.4.1 Multilayer spreading

Module `propagation_model`

What is propagation model?

In this library propagation model is considered as one or a plenty of phenomenas acting in one network, e.g. Suspected-Infected model.

Purpose of PropagationModel module

If experiment includes more than two phenomenas interacting with themselves, description of the propagation model becoming very complicated. E.g. model with 2 phenomenas with 2 local steps each:

- Suspected-Infected (phenomena Illness),
- Aware-Unaware (phenomena "Awareness"),

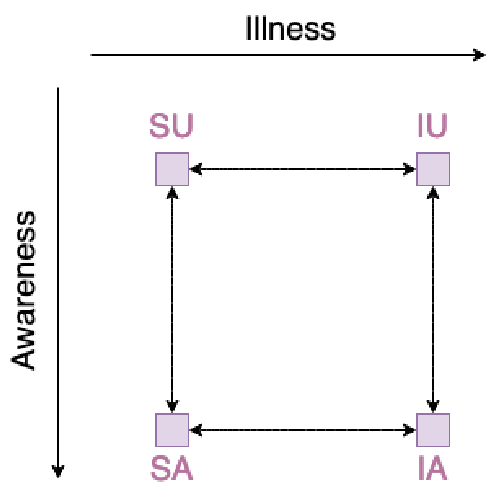
has 4 possible global states (i.e. for multiplex network each node has to be in one of those states):

- Suspected~Aware
- Suspected~Unaware,
- Infected~Aware,
- Infected~Unaware

and 8 possible transitions (i.e. possible ways for nodes in Multiplex network to change states):

- Suspected~Aware -> Suspected~Unaware,
- Suspected~Aware <- Suspected~Unaware,
- Infected~Aware -> Infected~Unaware,
- Infected~Aware <- Infected~Unaware,
- Suspected~Aware -> Infected~Aware,
- Suspected~Aware <- Infected~Aware,
- Suspected~Unaware -> Infected~Unaware,
- Suspected~Unaware <- Infected~Unaware.

This can be easily visualized by graph:



Note that with 3 phenomenas of respectively 2, 2, 3 local states we have 12 global states with (sic!) 48 possible transitions. This is so big value, that without computer assistance it is difficult to handle cases like this. Thus library contains module named `propagation_model` to define model in semi automatic way with no constrains coming from number od phenomenas and number of states. User defines names of phenomenas, local states and only these transitions which are relevant to the simulation.

Example of usage

Let's define model with 3 phenomenas, 2 (layer_1, layer_2) with 2 local states each (A, B) and 1 (layer_3) with 3 local states (A, B, C). Then assign probabilities of transitions between certain states.

Define object of model propagation:

```
from network_diffusion import PropagationModel
model = PropagationModel()
```

Assign phenomenas and local states. Then compile it ad see results:

```
model.add('layer_1', ('A', 'B'))
model.add('layer_2', ('A', 'B'))
model.add('layer_3', ('A', 'B', 'C'))
model.compile()
model.describe()
```

```
============================================
model of propagation
--------------------------------------------
phenomenas and their states:
layer_1: ('A', 'B')
layer_2: ('A', 'B')
layer_3: ('A', 'B', 'C')
background_weight: 0.0
layer 'layer_1' transitions with nonzero probability:
layer 'layer_2' transitions with nonzero probability:
layer 'layer_3' transitions with nonzero probability:
============================================
```

Assign nonzero probabilities to the propagation model code:

```
model.set_transition('layer_1', (('layer_1.A', 'layer_2.A', 'layer_3.A'),
    ('layer_1.B', 'layer_2.A', 'layer_3.A')), 0.5)
model.describe()
```

```
============================================
model of propagation
--------------------------------------------
phenomenas and their states:
    layer_1: ('A', 'B')
    layer_2: ('A', 'B')
    layer_3: ('A', 'B', 'C')
    background_weight: 0.0
layer 'layer_1' transitions with nonzero probability:
    from A to B with probability 0.5 and constrains ['layer_2.A' 'layer_3.A']
layer 'layer_2' transitions with nonzero probability:
layer 'layer_3' transitions with nonzero probability:
============================================
```

Set random transitions and see all model:

```
model.set_transitions_in_random_edges([[0.2, 0.3, 0.4], [0.2], [0.3]])
model.describe()
```

```
============================================
model of propagation
--------------------------------------------
phenomenas and their states:
    layer_1: ('A', 'B')
    layer_2: ('A', 'B')
    layer_3: ('A', 'B', 'C')
    background_weight: 0.0
layer 'layer_1' transitions with nonzero probability:
    from A to B with probability 0.2 and constrains ['layer_2.A' 'layer_3.A']
    from B to A with probability 0.3 and constrains ['layer_2.B' 'layer_3.A']
    from A to B with probability 0.4 and constrains ['layer_2.B' 'layer_3.C']
layer 'layer_2' transitions with nonzero probability:
    from A to B with probability 0.2 and constrains ['layer_1.B' 'layer_3.B']
layer 'layer_3' transitions with nonzero probability:
    from C to B with probability 0.3 and constrains ['layer_1.B' 'layer_2.B']
============================================
```

Because of the propagation model is stored as a dictionary of `networkx` graphs, user is able to draw it, but as the model is bigger as the readability of visualisation is less:

```
import matplotlib.pyplot as plt
for n, l in model.graph.items():
    plt.title(n)
    nx.draw_networkx_nodes(l, pos=nx.circular_layout(l))
    nx.draw_networkx_edges(l, pos=nx.circular_layout(l))
    nx.draw_networkx_edge_labels(l, pos=nx.circular_layout(l))
    nx.draw_networkx_labels(l, pos=nx.circular_layout(l))
    plt.show()
```

Module `multilayer_network`

What is a multilayer network?

Multilayer nNtwork is a class to extend functionality of `networkx.Graph` library to store and manipulate multilayer networks, which are a fundamental structure in the library. Module also allows to read network from *mpx* text files which stores such a structures.

Available data

Here is an exemplar repository with multilayer networks: hub, but you find them in many other sited around Internet.

Example of usage

Let’s crete some multilayer networks in several ways.

1. By defining separate graphs and layer names:

```
from network_diffusion.mln.mlnetwork import MultilayerNetwork
import networkx as nx

M = [nx.les_miserables_graph(), nx.les_miserables_graph(), nx.les_miserables_
→graph()]

mpx = MultilayerNetwork.from_nx_layers(M)
mpx.describe()
```

```
============================================
network parameters
--------------------------------------------
general parameters:
    number of layers: 3
    multiplexing coefficient: 1.0
layer 'layer_0' parameters:
    graph type - <class 'networkx.classes.graph.Graph'>
    number of nodes - 77
    number of edges - 254
    average degree - 6.5974
    clustering coefficient - 0.5731
layer 'layer_1' parameters:
    graph type - <class 'networkx.classes.graph.Graph'>
    number of nodes - 77
    number of edges - 254
    average degree - 6.5974
    clustering coefficient - 0.5731
layer 'layer_2' parameters:
    graph type - <class 'networkx.classes.graph.Graph'>
    number of nodes - 77
    number of edges - 254
    average degree - 6.5974
    clustering coefficient - 0.5731
============================================
```

2. By defining separate graphs and using default names of layers:

```
from network_diffusion.mln.mlnetwork import MultilayerNetwork
import networkx as nx
M = [nx.les_miserables_graph(), nx.les_miserables_graph(), nx.les_miserables_
→graph()]

mpx = MultilayerNetwork.from_nx_layer(M, ['A', 'B', 'C'])
mpx.describe()
```

```
============================================
network parameters
```

(continues on next page)

(continued from previous page)

```
--------------------------------------------
general parameters:
    number of layers: 3
    multiplexing coefficient: 1.0
layer 'A' parameters:
    graph type - <class 'networkx.classes.graph.Graph'>
    number of nodes - 77
    number of edges - 254
    average degree - 6.5974
    clustering coefficient - 0.5731
layer 'B' parameters:
    graph type - <class 'networkx.classes.graph.Graph'>
    number of nodes - 77
    number of edges - 254
    average degree - 6.5974
    clustering coefficient - 0.5731
layer 'C' parameters:
    graph type - <class 'networkx.classes.graph.Graph'>
    number of nodes - 77
    number of edges - 254
    average degree - 6.5974
    clustering coefficient - 0.5731
============================================
```

3. By reading out mpx file:

```
mpx = MultilayerNetwork.from_mpx('/my_project/monastery.mpx')
mpx.describe()
```

```
============================================
network parameters
--------------------------------------------
general parameters:
    number of layers: 10
    multiplexing coefficient: 0.7778
layer 'like1' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 18
    number of edges - 55
    average degree - 6.1111
    clustering coefficient - 0.1732
layer 'like2' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 18
    number of edges - 57
    average degree - 6.3333
    clustering coefficient - 0.2923
layer 'like3' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 18
    number of edges - 56
    average degree - 6.2222
```

(continues on next page)

(continued from previous page)

```
    clustering coefficient - 0.3603
layer 'dislike' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 17
    number of edges - 47
    average degree - 5.5294
    clustering coefficient - 0.1213
layer 'esteem' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 18
    number of edges - 54
    average degree - 6.0
    clustering coefficient - 0.3222
layer 'desesteem' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 17
    number of edges - 58
    average degree - 6.8235
    clustering coefficient - 0.2029
layer 'positive_influence' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 18
    number of edges - 53
    average degree - 5.8889
    clustering coefficient - 0.3537
layer 'negative_influence' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 18
    number of edges - 50
    average degree - 5.5556
    clustering coefficient - 0.1084
layer 'praise' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 18
    number of edges - 39
    average degree - 4.3333
    clustering coefficient - 0.3048
layer 'blame' parameters:
    graph type - <class 'networkx.classes.digraph.DiGraph'>
    number of nodes - 15
    number of edges - 41
    average degree - 5.4667
    clustering coefficient - 0.1133
============================================
```

Module simulator

How the simulator works?

Simulator is a class that allows to perform previously designed experiment. To run it we need a network (multilayer or temporal) (note that it can as well have one layer) and corresponding model. After the experiment is completed, user is able to see results in form of report and visualisation of global states of the nodes.

Example of usage

1. Initialise multilayer network from nx predefined network:

```
import networkx as nx
from network_diffusion.mln.mlnetwork import MultilayerNetwork

network = MultilayerNetwork()
names = ['illness', 'awareness', 'vaccination']
network.from_nx_layer(nx.les_miserables_graph(), names)
```

2. Initialise propagation model and set possible transitions with probabilities:

```
model = PropagationModel()
phenomenas = [('S', 'I', 'R'), ('UA', 'A'), ('UV', 'V')]
for l, p in zip(names, phenomenas):
    model.add(l, p)
model.compile(background_weight=0.005)

model.set_transition('illness.S', 'illness.I', ['vaccination.UV', 'awareness.UA'],␣
↪0.9)
model.set_transition('illness.S', 'illness.I', ['vaccination.V', 'awareness.A'], 0.
↪05)
model.set_transition('illness.S', 'illness.I', ['vaccination.UV', 'awareness.A'], 0.
↪2)
model.set_transition('illness.I', 'illness.R', ['vaccination.UV', 'awareness.UA'],␣
↪0.1)
model.set_transition('illness.I', 'illness.R', ['vaccination.V', 'awareness.A'], 0.
↪7)
model.set_transition('illness.I', 'illness.R', ['vaccination.UV', 'awareness.A'], 0.
↪3)

model.set_transition('vaccination.UV', 'vaccination.V', ['awareness.A', 'illness.S
↪'], 0.03)
model.set_transition('vaccination.UV', 'vaccination.V', ['awareness.A', 'illness.I
↪'], 0.01)

model.set_transition('awareness.UA', 'awareness.A', ['vaccination.UV', 'illness.S'],
↪ 0.05)
model.set_transition('awareness.UA', 'awareness.A', ['vaccination.V', 'illness.S'],␣
↪1)
model.set_transition('awareness.UA', 'awareness.A', ['vaccination.UV', 'illness.I'],
↪ 0.2)
```

(continues on next page)

(continued from previous page)

```
model.set_transition('illness', (('awareness.UA', 'illness.R', 'vaccination.UV'),
    ('awareness.UA', 'illness.I', 'vaccination.UV')), 0.7)
```

3. Initialise initial parameters of propagation in network. Parameters' names must correspond with names in model and network. Numbers in tuples describe how many nodes has which local state (in alphabetic order):

```
phenomenas = {'illness': (70, 6, 1), 'awareness': (60, 17), 'vaccination': (70, 7)}
```

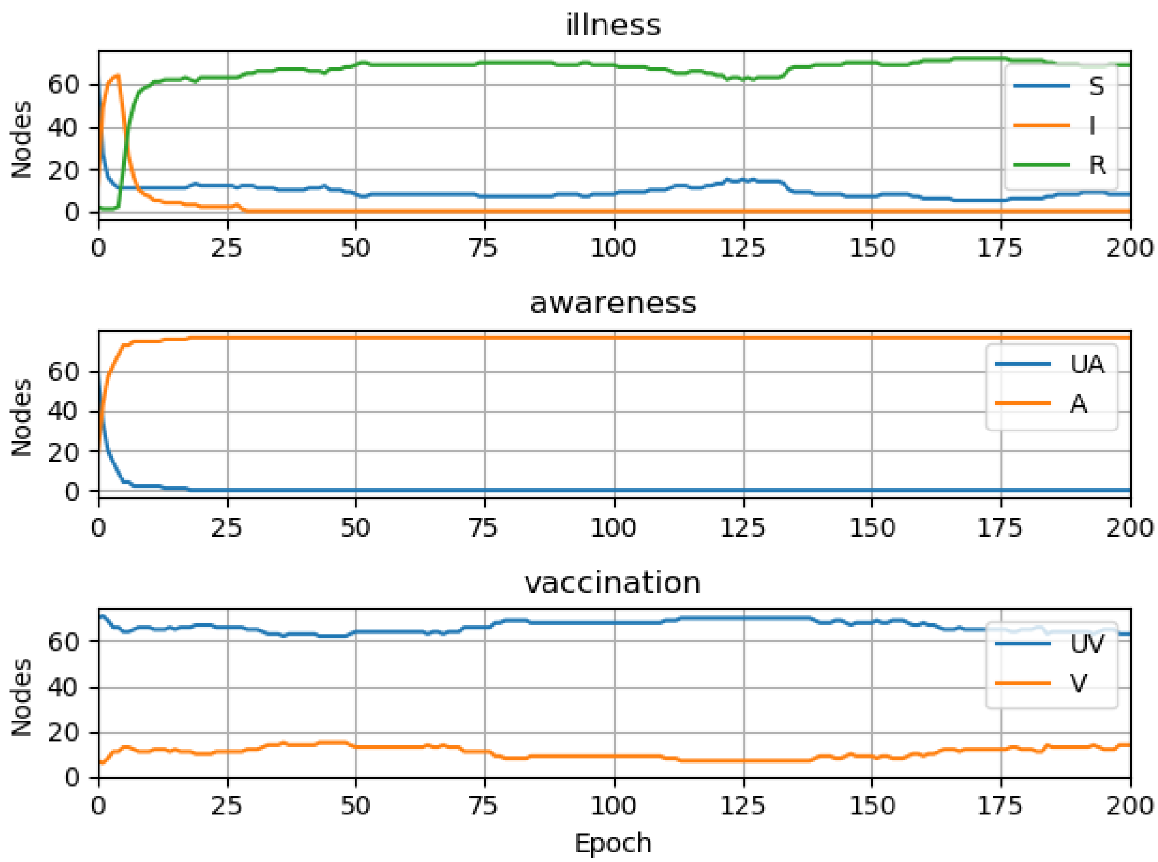
4. Perform propagation experiment. Propagation lasts as many epochs as defined (here 200). After the experiment, Logger object is returned where logs are being stored:

```
experiment = Simulator(model, network)
experiment.set_initial_states(phenomenas)
logs = experiment.perform_propagation(200)
```

5. Save experiment results. User is able to save them to file or print out to the console:

```
    logs.report(to_file=True, path=getcwd()+'/results', visualisation=True)

Logs contain:
    - description of the network (txt file)
    - description of the propagation model (txt file)
    - propagation report in all phenomena (separate csv file for each)
    - visualisation of propagation
```


illness
Nodes
S
I
R
0
20
40
60
0
25
50
75
100
125
150
175
200
awareness
Nodes
UA
A
vaccination
Nodes
UV
V
Epoch

CHAPTER

TWO

QUICK SEARCH

- genindex
- search

PYTHON MODULE INDEX

n

INDEX

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

P

R

S

T

U

V