FNSS core library

Release 0.8.2

Lorenzo Saino, Cosmin Cocora

Jul 29, 2018

Contents

This is the documentation of the FNSS core library. It is a Python library providing a set of features allowing to simplify the setup of a network experiment. These features include the ability to:

- Parse a topology from a dataset, a topology generator or generate it according to a number of synthetic models
- Apply link capacity, link weights, link delays and buffer sizes
- Deploy application stacks
- Generate traffic matrices
- Generate event schedules

The core library in addition to the features listed above, contains adapters to export generated scenarios to a the following network simulators or emulators: ns-2, Mininet, Omnet++, jFed and Autonetkit. Generated experiment scenarios (i.e. topologies, event schedules and traffic matrices) can be saved into XML files and then imported by libraries written in other languages. Currently, FNSS provides generic Java and C++ libraries as well as a C++ library specific for the ns-3 simulator. These libraries can be downloaded from the FNSS website.

The FNSS core library is released under the terms of the BSD license.

If you use FNSS for your paper, please cite the following publication:

> Lorenzo Saino, Cosmin Cocora and George Pavlou, A Toolchain for Symplifying Network Simulation Setup, in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques (SIMUTOOLS '13)*, Cannes, France, March 2013

The BibTeX entry is:

```
@inproceedings{fnss,
     author = {Saino, Lorenzo and Cocora, Cosmin and Pavlou, George},
     title = {A Toolchain for Simplifying Network Simulation Setup},
     booktitle = {Proceedings of the 6th International ICST Conference on Simulation
↪Tools and Techniques},
     series = {SIMUTOOLS '13},
     year = {2013},
     location = {Cannes, France},
     numpages = {10},
     publisher = {ICST (Institute for Computer Sciences, Social-Informatics and
↪Telecommunications Engineering)},
     address = {ICST, Brussels, Belgium, Belgium},
}
```

CHAPTER 1

Contents

1.1 Architecture

The Python core library is designed following a modular approach.

All functionalities are splitted in four main packages:

- **adapters**: contains functions for exporting FNSS objects to target simulators or emulators. Currently, this package includes functions for exporting FNSS objects to Mininet, ns-2, Omnet++, *jFed <http://jfed.iminds.be/>_* and AutoNetKit.
- **topologies**: contains all functions and classes for parsing or synthetically generating a network topology. It also contains functions to read and write topology objects from/to an XML file. The conversion of such objects to XML files is needed to make topology available for the Java and C++ API and the ns-3 adapter.
- **netconfig**: contains all functions for configuring a network topology. Such configuration include setting link capacities, delays and weights, set buffer sizes and deploy protocol stacks and applications on nodes.
- **traffic**: contains all functions and classes for synthetically generating event schedules and traffic matrices.

In addition, the library also comprises a set of classes to model specific entities. These classes are:

- **Topology**: a base undirected topology. Comprises methods for adding, editing and removing nodes and links. This class inherits from NetworkX Graph class. As a result, all graph algorithms and visualization tools provided by NetworkX can be used on Topology objects as well.
- **DirectedTopology**: a base directed topology. It shares most of the code of the Topology class but in this class links are directed. Similarly to the Topology class, this class inherits from NetworkX DiGraph class.
- **DatacenterTopology**: a datacenter topology. It inherits from the Topology class and comprises additional methods relevant only for datacenter topologies.
- **TrafficMatrix**: a traffic matrix, capturing the average traffic on a network at a specific point in time.
- **TrafficMatrixSequence**: a sequence of traffic matrices, capturing the evolution of traffic on a network over a period of time.
- **EventSchedule**: a schedule of events to be simulated.

In order to make the simulation setup information created with FNSS core library (topology, traffic, events) available to the desired target simulator, FNSS provides the capability to export such information to XML files. These XML files can then be read by the Java, C++ or ns-3 libraries. More specifically, the following objects can be saved to XML files:

- **Topology**, **DirectedTopology**, **DatacenterTopology** and any potential subclasses can be written to XML files with the function `write_topology`.
- **TrafficMatrix**, **TrafficMatrixSequence** and any potential subclasses can be written to XML files with the function `write_traffic_matrix`.
- **EventSchedule** and any potential subclasses can be written to XML files with the function `write_event_schedule`.

1.2 Install

1.2.1 Quick install

1.2.1.1 Ubuntu (version 12.04+)

If you use Ubuntu, you can install the FNSS core library along with the Python interpreter and all required dependencies by running the following script:

```
$ curl -L https://github.com/fnss/fnss/raw/master/core/ubuntu_install.sh | sh
```

You need superuser privileges to run this script.

1.2.1.2 Other operating systems

The easiest way to install the core Python library is to download it and install it from the Python Package Index. To do so, you must have Python (version >= 2.7) installed on your machine and either *pip* or *easy_install*.

To install the FNSS core library using *easy_install* open a command shell and type:

```
$ easy_install fnss
```

If you use *pip*, type instead:

```
$ pip fnss
```

Depending on the configuration of your machine you may need to run *pip* or *easy_install* as superuser. Whether you use *pip* or *easy_install*, the commands reported above will download the latest version of the FNSS core library and install it on your machine together with all required dependencies.

1.2.2 Installing from source

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

1.2.2.1 Source archive file

1. Download the source (tar.gz or zip file) from http://fnss.github.io
2. Unpack, open a command shell and move to the main directory of the core library (it should have the file *setup.py*).
3. Run this instruction to build and install:

```
$ python setup.py install
```

1.2.2.2 Git repository

1. Clone the FNSS repostitory:

```
$ git clone https://github.com/fnss/fnss.git
```

2. Change directory to *fnss/core*:

```
$ cd fnss/core
```

3. Run:

```
$ python setup.py 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/inst/search-path.html for further details.

1.2.3 Requirements

1.2.3.1 Python

To use FNSS you need Python version 2.7 or later. FNSS fully supports Python 3

1.2.3.2 Required packages

The following packages are needed by FNSS to provide core functions.

NetworkX (version >= 1.6)

NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

- Download: http://networkx.github.io

NumPy (version >= 1.4)

Provides matrix representation of graphs and is used in some graph algorithms for high-performance matrix computations.

- Download: http://scipy.org/Download

Mako (version >= 0.4)

It is a templating engine used to export FNSS topologies.

- Download: http://www.makotemplates.org/download.html

1.3 API Reference

1.3.1 Classes

1.3.1.1 Topology

class Topology(*data=None*, *name=''*, ***kwargs*)

Base class for undirected topology

Attributes

name

Methods

`add_cycle`(nodes, **attr)	Add a cycle.
`add_edge`(u, v[, attr_dict])	Add an edge between u and v.
`add_edges_from`(ebunch[, attr_dict])	Add all the edges in ebunch.
`add_node`(n[, attr_dict])	Add a single node n and update node attributes.
`add_nodes_from`(nodes, **attr)	Add multiple nodes.
`add_path`(nodes, **attr)	Add a path.
`add_star`(nodes, **attr)	Add a star.
`add_weighted_edges_from`(ebunch[, weight])	Add all the edges in ebunch as weighted edges with specified weights.
`adjacency_iter`()	Return an iterator of (node, adjacency dict) tuples for all nodes.
`adjacency_list`()	Return an adjacency list representation of the graph.
`adjlist_dict_factory`	alias of `__builtin__.dict`

Continued on next page

Table 1 – continued from previous page

applications()	Return a dictionary of all applications deployed, keyed by node
buffers()	Return a dictionary of all buffer sizes, keyed by interface
capacities()	Return a dictionary of all link capacities, keyed by link
clear()	Remove all nodes and edges from the graph.
copy()	Return a copy of the topology.
degree([nbunch, weight])	Return the degree of a node or nodes.
degree_iter([nbunch, weight])	Return an iterator for (node, degree).
delays()	Return a dictionary of all link delays, keyed by link
edge_attr_dict_factory	alias of `__builtin__.dict`
edges([nbunch, data, default])	Return a list of edges.
edges_iter([nbunch, data, default])	Return an iterator over the edges.
get_edge_data(u, v[, default])	Return the attribute dictionary associated with edge (u,v).
has_edge(u, v)	Return True if the edge (u,v) is in the graph.
has_node(n)	Return True if the graph contains the node n.
is_directed()	Return True if graph is directed, False otherwise.
is_multigraph()	Return True if graph is a multigraph, False otherwise.
nbunch_iter([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.
neighbors(n)	Return a list of the nodes connected to the node n.
neighbors_iter(n)	Return an iterator over all neighbors of node n.
node_dict_factory	alias of `__builtin__.dict`
nodes([data])	Return a list of the nodes in the graph.
nodes_iter([data])	Return an iterator over the nodes.
nodes_with_selfloops()	Return a list of nodes with self loops.
number_of_edges([u, v])	Return the number of edges between two nodes.
number_of_nodes()	Return the number of nodes in the graph.
number_of_selfloops()	Return the number of selfloop edges.
order()	Return the number of nodes in the graph.
remove_edge(u, v)	Remove the edge between u and v.
remove_edges_from(ebunch)	Remove all edges specified in ebunch.
remove_node(n)	Remove node n.
remove_nodes_from(nodes)	Remove multiple nodes.
selfloop_edges([data, default])	Return a list of selfloop edges.
size([weight])	Return the number of edges.
stacks()	Return a dictionary of all node stacks, keyed by node
subgraph(nbunch)	Return the subgraph induced on nodes in nbunch.
to_directed()	Return a directed representation of the topology.
to_undirected()	Return an undirected copy of the topology.
weights()	Return a dictionary of all link weights, keyed by link

---

1.3.1.2 DirectedTopology

class DirectedTopology (*data=None*, *name=''*, ***kwargs*)
Base class for directed topology

Attributes

name

Methods

`add_cycle`(nodes, **attr)	Add a cycle.
`add_edge`(u, v[, attr_dict])	Add an edge between u and v.
`add_edges_from`(ebunch[, attr_dict])	Add all the edges in ebunch.
`add_node`(n[, attr_dict])	Add a single node n and update node attributes.
`add_nodes_from`(nodes, **attr)	Add multiple nodes.
`add_path`(nodes, **attr)	Add a path.
`add_star`(nodes, **attr)	Add a star.
`add_weighted_edges_from`(ebunch[, weight])	Add all the edges in ebunch as weighted edges with specified weights.
`adjacency_iter`()	Return an iterator of (node, adjacency dict) tuples for all nodes.
`adjacency_list`()	Return an adjacency list representation of the graph.
`adjlist_dict_factory`	alias of `__builtin__.dict`
`applications`()	Return a dictionary of all applications deployed, keyed by node
`buffers`()	Return a dictionary of all buffer sizes, keyed by interface
`capacities`()	Return a dictionary of all link capacities, keyed by link
`clear`()	Remove all nodes and edges from the graph.
`copy`()	Return a copy of the topology.
`degree`([nbunch, weight])	Return the degree of a node or nodes.
`degree_iter`([nbunch, weight])	Return an iterator for (node, degree).
`delays`()	Return a dictionary of all link delays, keyed by link
`edge_attr_dict_factory`	alias of `__builtin__.dict`
`edges`([nbunch, data, default])	Return a list of edges.
`edges_iter`([nbunch, data, default])	Return an iterator over the edges.
`get_edge_data`(u, v[, default])	Return the attribute dictionary associated with edge (u,v).
`has_edge`(u, v)	Return True if the edge (u,v) is in the graph.
`has_node`(n)	Return True if the graph contains the node n.
`has_predecessor`(u, v)	Return True if node u has predecessor v.
`has_successor`(u, v)	Return True if node u has successor v.
`in_degree`([nbunch, weight])	Return the in-degree of a node or nodes.
`in_degree_iter`([nbunch, weight])	Return an iterator for (node, in-degree).
`in_edges`([nbunch, data])	Return a list of the incoming edges.
`in_edges_iter`([nbunch, data])	Return an iterator over the incoming edges.
`is_directed`()	Return True if graph is directed, False otherwise.
`is_multigraph`()	Return True if graph is a multigraph, False otherwise.
`nbunch_iter`([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.
`neighbors`(n)	Return a list of successor nodes of n.
`neighbors_iter`(n)	Return an iterator over successor nodes of n.

Continued on next page

Table 3 – continued from previous page

`node_dict_factory`	alias of `__builtin__.dict`
`nodes`([data])	Return a list of the nodes in the graph.
`nodes_iter`([data])	Return an iterator over the nodes.
`nodes_with_selfloops`()	Return a list of nodes with self loops.
`number_of_edges`([u, v])	Return the number of edges between two nodes.
`number_of_nodes`()	Return the number of nodes in the graph.
`number_of_selfloops`()	Return the number of selfloop edges.
`order`()	Return the number of nodes in the graph.
`out_degree`([nbunch, weight])	Return the out-degree of a node or nodes.
`out_degree_iter`([nbunch, weight])	Return an iterator for (node, out-degree).
`out_edges`([nbunch, data, default])	Return a list of edges.
`out_edges_iter`([nbunch, data, default])	Return an iterator over the edges.
`predecessors`(n)	Return a list of predecessor nodes of n.
`predecessors_iter`(n)	Return an iterator over predecessor nodes of n.
`remove_edge`(u, v)	Remove the edge between u and v.
`remove_edges_from`(ebunch)	Remove all edges specified in ebunch.
`remove_node`(n)	Remove node n.
`remove_nodes_from`(nbunch)	Remove multiple nodes.
`reverse`([copy])	Return the reverse of the graph.
`selfloop_edges`([data, default])	Return a list of selfloop edges.
`size`([weight])	Return the number of edges.
`stacks`()	Return a dictionary of all node stacks, keyed by node
`subgraph`(nbunch)	Return the subgraph induced on nodes in nbunch.
`successors`(n)	Return a list of successor nodes of n.
`successors_iter`(n)	Return an iterator over successor nodes of n.
`to_directed`()	Return a directed representation of the topology.
`to_undirected`()	Return an undirected copy of the topology.
`weights`()	Return a dictionary of all link weights, keyed by link

1.3.1.3 DatacenterTopology

class DatacenterTopology(*data=None*, *name=''*, ***kwargs*)

Represent a datacenter topology

Attributes

name

Methods

`add_cycle`(nodes, **attr)	Add a cycle.
`add_edge`(u, v[, attr_dict])	Add an edge between u and v.
`add_edges_from`(ebunch[, attr_dict])	Add all the edges in ebunch.
`add_node`(n[, attr_dict])	Add a single node n and update node attributes.
`add_nodes_from`(nodes, **attr)	Add multiple nodes.
`add_path`(nodes, **attr)	Add a path.
`add_star`(nodes, **attr)	Add a star.

Continued on next page

Table 5 – continued from previous page

add_weighted_edges_from(ebunch[, weight])	Add all the edges in ebunch as weighted edges with specified weights.
adjacency_iter()	Return an iterator of (node, adjacency dict) tuples for all nodes.
adjacency_list()	Return an adjacency list representation of the graph.
adjlist_dict_factory	alias of __builtin__.dict
applications()	Return a dictionary of all applications deployed, keyed by node
buffers()	Return a dictionary of all buffer sizes, keyed by interface
capacities()	Return a dictionary of all link capacities, keyed by link
clear()	Remove all nodes and edges from the graph.
copy()	Return a copy of the topology.
degree([nbunch, weight])	Return the degree of a node or nodes.
degree_iter([nbunch, weight])	Return an iterator for (node, degree).
delays()	Return a dictionary of all link delays, keyed by link
edge_attr_dict_factory	alias of __builtin__.dict
edges([nbunch, data, default])	Return a list of edges.
edges_iter([nbunch, data, default])	Return an iterator over the edges.
get_edge_data(u, v[, default])	Return the attribute dictionary associated with edge (u,v).
has_edge(u, v)	Return True if the edge (u,v) is in the graph.
has_node(n)	Return True if the graph contains the node n.
hosts()	Return the list of host nodes in the topology
is_directed()	Return True if graph is directed, False otherwise.
is_multigraph()	Return True if graph is a multigraph, False otherwise.
nbunch_iter([nbunch])	Return an iterator of nodes contained in nbunch that are also in the graph.
neighbors(n)	Return a list of the nodes connected to the node n.
neighbors_iter(n)	Return an iterator over all neighbors of node n.
node_dict_factory	alias of __builtin__.dict
nodes([data])	Return a list of the nodes in the graph.
nodes_iter([data])	Return an iterator over the nodes.
nodes_with_selfloops()	Return a list of nodes with self loops.
number_of_edges([u, v])	Return the number of edges between two nodes.
number_of_hosts()	Return the number of hosts in the topology
number_of_nodes()	Return the number of nodes in the graph.
number_of_selfloops()	Return the number of selfloop edges.
number_of_switches()	Return the number of switches in the topology
order()	Return the number of nodes in the graph.
remove_edge(u, v)	Remove the edge between u and v.
remove_edges_from(ebunch)	Remove all edges specified in ebunch.
remove_node(n)	Remove node n.
remove_nodes_from(nodes)	Remove multiple nodes.
selfloop_edges([data, default])	Return a list of selfloop edges.
size([weight])	Return the number of edges.
stacks()	Return a dictionary of all node stacks, keyed by node
subgraph(nbunch)	Return the subgraph induced on nodes in nbunch.
switches()	Return the list of switch nodes in the topology

Continued on next page

Table 5 – continued from previous page

to_directed()	Return a directed representation of the topology.
to_undirected()	Return an undirected copy of the topology.
weights()	Return a dictionary of all link weights, keyed by link

1.3.1.4 TrafficMatrix

class TrafficMatrix(*volume_unit='Mbps'*, *flows=None*)

Class representing a single traffic matrix.

It simply contains a set of traffic volumes being exchanged between origin-destination pairs

Parameters

volume_unit [str] The unit in which traffic volumes are expressed

flows [dict, optional] The traffic volumes or the matrix, keyed by origin-destination pair. The origin-destination pair is a tuple whose two elements are respectively the identifier of the origin and destination nodes and volumes are all expressed in the same unit

Methods

add_flow(origin, destination, volume)	Add a flow to the traffic matrix
flows()	Return the flows of the traffic matrix
od_pairs()	Return all OD pairs of the traffic matrix
pop_flow(origin, destination)	Pop a flow from the traffic matrix and return the volume of the flow removed.

1.3.1.5 TrafficMatrixSequence

class TrafficMatrixSequence(*interval=None*, *t_unit='min'*)

Class representing a sequence of traffic matrices.

Parameters

interval [float or int, optional] The time interval elapsed between subsequent traffic matrices of the sequence

t_unit [str, optional] The unit of the interval value (e.g. 'sec' or 'min')

Methods

append(tm)	Append a traffic matrix at the end of the sequence
get(i)	Return a specific traffic matrix in a specific position of the sequence
insert(i, tm)	Insert a traffic matrix in the sequence at a specified position

Continued on next page

Table 9 – continued from previous page

pop(i)	Removes the traffic matrix in a specific position of the sequence

1.3.1.6 EventSchedule

class EventSchedule (*t_start=0, t_unit='ms'*)

Class representing an event schedule. This class is simply a wrapper for a list of events.

Methods

add(time, event[, absolute_time])	Adds an event to the schedule.
add_schedule(event_schedule)	Merge with another event schedule.
events_between(t_start, t_end)	Return an event schedule comprising all events scheduled between a start time (included) and an end time (excluded).
number_of_events()	Return the number of events in the schedule
pop(i)	Remove from the schedule the event in a specific position

1.3.2 Functions

1.3.2.1 netconfig package

buffers module

Function to assign and manipulate buffer sizes of network interfaces.

clear_buffer_sizes(topology)	Remove all buffer sizes from the topology.
get_buffer_sizes(topology)	Returns all the buffer sizes.
set_buffer_sizes_bw_delay_prod(topology[, ...])	Assign a buffer sizes proportionally to the product of link bandwidth and average network RTT.
set_buffer_sizes_constant(topology, buffer_size)	Assign a constant buffer size to all selected interfaces
set_buffer_sizes_link_bandwidth(topology[, ...])	Assign a buffer sizes proportionally to the bandwidth of the interface on which the flush.

fnss.netconfig.buffers.clear_buffer_sizes

clear_buffer_sizes (*topology*)

Remove all buffer sizes from the topology.

Parameters

topology [Topology or DirectedTopology] The topology whose buffer sizes are cleared

fnss.netconfig.buffers.get_buffer_sizes

get_buffer_sizes(*topology*)

Returns all the buffer sizes.

Parameters

topology [Topology or DirectedTopology]

Returns

buffer_sizes [dict] Dictionary of buffer sizes keyed by (u, v) tuple. The key (u, v) represents a network interface where u is the node on which the interface is located and (u, v) is the link to which the buffer flushes

Examples

```
>>> import fnss
>>> topology = fnss.Topology()
>>> topology.add_path([1, 2, 3])
>>> fnss.set_buffer_sizes_constant(topology, buffer_size=10)
>>> buffer = fnss.get_buffer_sizes(topology)
>>> buffer[(1,2)]
10
```

fnss.netconfig.buffers.set_buffer_sizes_bw_delay_prod

set_buffer_sizes_bw_delay_prod(*topology*, *buffer_unit='bytes'*, *packet_size=1500*)

Assign a buffer sizes proportionally to the product of link bandwidth and average network RTT. This is a rule of thumb according to which the buffers of Internet routers are generally configured.

Parameters

topology [Topology or DirectedTopology] The topology on which delays are applied.

buffer_unit [string] The unit of buffer sizes. Supported units are: *bytes* and *packets*

packet_size [int, optional] The average packet size (in bytes). It used only if *packets* is selected as buffer size to properly calculate buffer sizes given bandwidth and delay values.

Examples

```
>>> import fnss
>>> topology = fnss.erdos_renyi_topology(50, 0.2)
>>> fnss.set_capacities_constant(topology, 10, 'Mbps')
>>> fnss.set_delays_constant(topology, 2, 'ms')
>>> fnss.set_buffer_sizes_bw_delay_prod(topology)
```

fnss.netconfig.buffers.set_buffer_sizes_constant

set_buffer_sizes_constant(*topology*, *buffer_size*, *buffer_unit='bytes'*, *interfaces=None*)

Assign a constant buffer size to all selected interfaces

Parameters

topology [Topology or DirectedTopology] The topology on which buffer sizes are applied.

buffer_size [int] The constant buffer_size to be applied to all interface

buffer_unit [string, unit] The unit of buffer sizes. Supported units are: *bytes* and *packets*

interfaces [iterable container of tuples, optional] Iterable container of selected interfaces on which buffer sizes are applied. An interface is defined by the tuple (u,v) where u is the node on which the interface is located and (u,v) is the link to which the buffer flushes.

Examples

```
>>> import fnss
>>> topology = fnss.Topology()
>>> topology.add_path([1, 2, 4, 5, 8])
>>> fnss.set_buffer_sizes_constant(topology, 100000, buffer_unit='bytes',     ...␣
↪interfaces=[(1,2), (5,8), (4,5)])
```

fnss.netconfig.buffers.set_buffer_sizes_link_bandwidth

set_buffer_sizes_link_bandwidth(*topology*, *k=1.0*, *default_size=None*, *buffer_unit='bytes'*, *packet_size=1500*)

Assign a buffer sizes proportionally to the bandwidth of the interface on which the flush. In particularly, the buffer size will be equal to $kimesC$, where C is the capacity of the link in bps.

This assignment is equal to the bandwidth-delay product if k is the average RTT in seconds.

To use this function, all links of the topology must have a *capacity* attribute. If the length of a link cannot be determined, it is applied the delay equal *default_delay* if specified, otherwise an error is returned.

Parameters

topology [Topology or DirectedTopology] The topology on which delays are applied.

k [float, optional] The multiplicative constant applied to capacity to derive buffer size

default_size [float, optional] The buffer size to be applied to interfaces whose speed is unknown. If it is None and at least one link does not have a capacity attribute, return an error

buffer_unit [string, unit] The unit of buffer sizes. Supported units are: *bytes* and *packets*

packet_size [int, optional] The average packet size (in bytes). It used only if *packets* is selected as buffer size to properly calculate buffer sizes given bandwidth and delay values.

Examples

```
>>> import fnss
>>> topology = fnss.erdos_renyi_topology(50, 0.1)
>>> fnss.set_capacities_constant(topology, 10, 'Mbps')
>>> fnss.set_delays_constant(topology, 2, 'ms')
>>> fnss.set_buffer_sizes_link_bandwidth(topology, k=1.0)
```

capacities module

Functions to assign and manipulate link capacities of a topology.

Link capacities can be assigned either deterministically or randomly, according to various models.

clear_capacities(topology)	Remove all capacities from the topology.
get_capacities(topology)	Returns a dictionary with all link capacities.
set_capacities_betweenness_gravity(topology, …)	Set link capacities proportionally to the product of the betweenness centralities of the two end-points of the link
set_capacities_communicability_gravity(…)	Set link capacities proportionally to the product of the communicability centralities of the two end-points of the link
set_capacities_constant(topology, capacity)	Set constant link capacities
set_capacities_degree_gravity(topology, …)	Set link capacities proportionally to the product of the degrees of the two end-points of the link
set_capacities_edge_betweenness(topology, …)	Set link capacities proportionally to edge betweenness centrality of the link.
set_capacities_edge_communicability(…[, …])	Set link capacities proportionally to edge communicability centrality of the link.
set_capacities_eigenvector_gravity(topology, …)	Set link capacities proportionally to the product of the eigenvector centralities of the two end-points of the link
set_capacities_pagerank_gravity(topology, …)	Set link capacities proportionally to the product of the Pagerank centralities of the two end-points of the link
set_capacities_random(topology, capacity_pdf)	Set random link capacities according to a given probability density function
set_capacities_random_power_law(topology, …)	Set random link capacities according to a power-law probability density function.
set_capacities_random_uniform(topology, …)	Set random link capacities according to a uniform probability density function.
set_capacities_random_zipf(topology, capacities)	Set random link capacities according to a Zipf probability density function.
set_capacities_random_zipf_mandelbrot(…[, …])	Set random link capacities according to a Zipf-Mandelbrot probability density function.

fnss.netconfig.capacities.clear_capacities

clear_capacities(*topology*)

Remove all capacities from the topology.

Parameters

topology [Topology]

fnss.netconfig.capacities.get_capacities

get_capacities(*topology*)

Returns a dictionary with all link capacities.

Parameters

topology [Topology] The topology whose link delays are requested

Returns

capacities [dict] Dictionary of link capacities keyed by link.

Examples

```
>>> import fnss
>>> topology = fnss.Topology()
>>> topology.add_path([1,2,3])
>>> fnss.set_capacities_constant(topology, 10, 'Mbps')
>>> capacity = get_capacities(topology)
>>> capacity[(1,2)]
10
```

fnss.netconfig.capacities.set_capacities_betweenness_gravity

set_capacities_betweenness_gravity(*topology*, *capacities*, *capacity_unit='Mbps'*, *weighted=True*)

Set link capacities proportionally to the product of the betweenness centralities of the two end-points of the link

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

weighted [bool, optional] Indicate whether link weights need to be used to compute shortest paths. If links do not have link weights or this parameter is False, shortest paths are calculated based on hop count.

fnss.netconfig.capacities.set_capacities_communicability_gravity

set_capacities_communicability_gravity(*topology*, *capacities*, *capacity_unit='Mbps'*)

Set link capacities proportionally to the product of the communicability centralities of the two end-points of the link

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

fnss.netconfig.capacities.set_capacities_constant

set_capacities_constant(*topology*, *capacity*, *capacity_unit='Mbps'*, *links=None*)

Set constant link capacities

Parameters

topology [Topology] The topology to which link capacities will be set

capacity [float] The value of capacity to set

links [iterable, optional] Iterable container of links, represented as (u, v) tuples to which capacity will be set. If None or not specified, the capacity will be applied to all links.

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

Examples

```
>>> import fnss
>>> topology = fnss.erdos_renyi_topology(50, 0.1)
>>> fnss.set_capacities_constant(topology, 10, 'Mbps')
```

fnss.netconfig.capacities.set_capacities_degree_gravity

set_capacities_degree_gravity(*topology*, *capacities*, *capacity_unit='Mbps'*)
Set link capacities proportionally to the product of the degrees of the two end-points of the link

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

fnss.netconfig.capacities.set_capacities_edge_betweenness

set_capacities_edge_betweenness(*topology*, *capacities*, *capacity_unit='Mbps'*, *weighted=True*)
Set link capacities proportionally to edge betweenness centrality of the link.

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

weighted [bool, optional] Indicate whether link weights need to be used to compute shortest paths. If links do not have link weights or this parameter is False, shortest paths are calculated based on hop count.

fnss.netconfig.capacities.set_capacities_edge_communicability

set_capacities_edge_communicability(*topology*, *capacities*, *capacity_unit='Mbps'*)
Set link capacities proportionally to edge communicability centrality of the link.

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

fnss.netconfig.capacities.set_capacities_eigenvector_gravity

set_capacities_eigenvector_gravity(*topology*, *capacities*, *capacity_unit='Mbps'*, *max_iter=1000*)
Set link capacities proportionally to the product of the eigenvector centralities of the two end-points of the link

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

max_iter [int, optional] The max number of iteration of the algorithm allowed. If a solution is not found within this period

Raises

RuntimeError [if the algorithm does not converge in max_iter iterations]

fnss.netconfig.capacities.set_capacities_pagerank_gravity

set_capacities_pagerank_gravity(*topology*, *capacities*, *capacity_unit='Mbps'*, *alpha=0.85*, *weight=None*)
Set link capacities proportionally to the product of the Pagerank centralities of the two end-points of the link

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

alpha [float, optional] The apha parameter of the PageRank algorithm

weight [str, optional] The name of the link attribute to use for the PageRank algorithm. Valid attributes include *capacity delay* and *weight*. If `None`, all links are assigned the same weight.

fnss.netconfig.capacities.set_capacities_random

set_capacities_random(*topology*, *capacity_pdf*, *capacity_unit='Mbps'*)
Set random link capacities according to a given probability density function

Parameters

topology [Topology] The topology to which link capacities will be set

capacity_pdf [dict] A dictionary representing the probability that a capacity value is assigned to a link

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

links [list, optional] List of links, represented as (u, v) tuples to which capacity will be set. If None or not specified, the capacity will be applied to all links.

Examples

```
>>> import fnss
>>> topology = fnss.erdos_renyi_topology(50, 0.1)
>>> pdf = {10: 0.5, 100: 0.2, 1000: 0.3}
>>> fnss.set_capacities_constant(topology, pdf, 'Mbps')
```

fnss.netconfig.capacities.set_capacities_random_power_law

set_capacities_random_power_law(*topology*, *capacities*, *capacity_unit='Mbps'*, *alpha=1.1*)
Set random link capacities according to a power-law probability density function.

The probability that a capacity c_i is assigned to a link is:

$$p(c_i) = \frac{{c_i}^{-\alpha}}{\sum_{c_k \in C}{c_k}^{-\alpha}}.$$

Where C is the set of allowed capacity, i.e. the `capacities` argument

Note that this capacity assignment differs from `set_capacities_random_zipf` because, while in Zipf assignment the power law relationship is between the rank of a capacity and the probability of being assigned to a link, in this assignment, the power law is between the value of the capacity and the probability of being assigned to a link.

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

fnss.netconfig.capacities.set_capacities_random_uniform

set_capacities_random_uniform(*topology*, *capacities*, *capacity_unit='Mbps'*)
Set random link capacities according to a uniform probability density function.

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

fnss.netconfig.capacities.set_capacities_random_zipf

set_capacities_random_zipf(*topology*, *capacities*, *capacity_unit='Mbps'*, *alpha=1.1*, *reverse=False*)
Set random link capacities according to a Zipf probability density function.

The same objective can be achieved by invoking the function `set_capacities_random_zipf_mandlebrot` with parameter q set to 0.

This capacity allocation consists in the following steps:

1. All capacities are sorted in descending or order (or ascending if reverse is True)
2. The i-th value of the sorted capacities list is then assigned to a link with probability

$$p(i) = \frac{1/i^\alpha}{\sum_{i=1}^{N} 1/i^\alpha}.$$

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

alpha [float, default 1.1] The :math'lpha' parameter of the Zipf density function

reverse [bool, optional] If False, lower capacity links are the most frequent, if True, higher capacity links are more frequent

fnss.netconfig.capacities.set_capacities_random_zipf_mandelbrot

`set_capacities_random_zipf_mandelbrot`(*topology*, *capacities*, *capacity_unit='Mbps'*, *alpha=1.1*, *q=0.0*, *reverse=False*)

Set random link capacities according to a Zipf-Mandelbrot probability density function.

This capacity allocation consists in the following steps:

1. All capacities are sorted in descending or order (or ascending if reverse is True)
2. The i-th value of the sorted capacities list is then assigned to a link with probability

$$p(i) = \frac{1/(i+q)^\alpha}{\sum_{i=1}^{N} 1/(i+q)^\alpha}.$$

Parameters

topology [Topology] The topology to which link capacities will be set

capacities [list] A list of all possible capacity values

capacity_unit [str, optional] The unit in which capacity value is expressed (e.g. Mbps, Gbps etc..)

alpha [float, default 1.1] The :math'lpha' parameter of the Zipf-Mandlebrot density function

q [float, default 0] The :math'q' parameter of the Zipf-Mandlebrot density function

reverse [bool, optional] If False, lower capacity links are the most frequent, if True, higher capacity links are more frequent

`delays` module

Functions to assign and manipulate link delays.

clear_delays(topology)	Remove all delays from the topology.
get_delays(topology)	Returns all the delays.
set_delays_constant(topology[, delay, …])	Assign a constant delay to all selected links
set_delays_geo_distance(topology, specific_delay)	Assign a delay to all selected links equal to the product of link length and specific delay.

fnss.netconfig.delays.clear_delays

clear_delays (*topology*)

Remove all delays from the topology.

Parameters

topology [Topology]

fnss.netconfig.delays.get_delays

get_delays (*topology*)

Returns all the delays.

Parameters

topology [Topology] The topology whose link delays are requested

Returns

delays [dict] Dictionary of link delays keyed by link.

Examples

```
>>> import fnss
>>> topology = fnss.Topology()
>>> topology.add_path([1,2,3])
>>> fnss.set_delays_constant(topology, 10, 'ms')
>>> delay = get_delays(topology)
>>> delay[(1,2)]
10
```

fnss.netconfig.delays.set_delays_constant

set_delays_constant (*topology*, *delay=1.0*, *delay_unit='ms'*, *links=None*)

Assign a constant delay to all selected links

Parameters

topology [Topology] The topology on which delays are applied.

delay [float, optional] The constant delay to be applied to all links

delay_unit [string, optional] The unit of delays. Supported units are: "us" (microseconds), "ms" (milliseconds) and "s" (seconds)

links [list, optional] List of selected links on which weights are applied. If it is None, all links are selected

Examples

```
>>> import fnss
>>> topology = fnss.Topology()
>>> topology.add_path([1, 2, 4, 5, 8])
>>> fnss.set_delays_constant(topology, 5.0, 'ms', links=[(1,2), (5,8), (4,5)])
>>> delay = fnss.get_delays(topology)
>>> delay[(1, 2)]
5.0
```

fnss.netconfig.delays.set_delays_geo_distance

set_delays_geo_distance(*topology*, *specific_delay*, *default_delay=None*, *delay_unit='ms'*, *links=None*)

Assign a delay to all selected links equal to the product of link length and specific delay. To use this function, all nodes must have a 'latitude' and a 'longitude' attribute. Alternatively, all links of the topology must have a 'length' attribute. If the length of a link cannot be determined, it is applied the delay equal default_delay if specified, otherwise an error is returned.

Parameters

topology [Topology] The topology on which delays are applied.

specific_delay [float] The specific delay (in ms/Km) to be applied to all links

default_delay [float, optional] The delay to be applied to links whose length is not known. If None, if the length of a link cannot be determined, an error is returned

delay_unit [string, optional] The unit of delays. Supported units are: "us" (microseconds), "ms" (milliseconds) and "s" (seconds)

links [list, optional] List of selected links on which weights are applied. If it is None, all links are selected

Examples

```
>>> import fnss
>>> topology = fnss.parse_abilene('abilene_topo.txt')
>>> fnss.set_delays_geo_distance(topology, specific_delay=fnss.PROPAGATION_DELAY_
↪FIBER)
```

nodeconfig module

Functions to deploy and configure protocol stacks and applications on network nodes

add_application(topology, node, name[, …])	Add an application to a node
add_stack(topology, node, name[, properties])	Set stack on a node.
clear_applications(topology)	Remove all applications from all nodes of the topology
clear_stacks(topology)	Remove all stacks from all nodes of the topology
get_application_names(topology, node)	Return a list of names of applications deployed on a node

Continued on next page

Table 16 – continued from previous page

get_application_properties(topology, node, name)	Return a dictionary containing all the properties of an application deployed on a node
get_stack(topology, node[, data])	Return the stack of a node, if any
remove_application(topology, node[, name])	Remove an application from a node
remove_stack(topology, node)	Remove stack from a node

fnss.netconfig.nodeconfig.add_application

add_application (*topology*, *node*, *name*, *properties=None*, ***attr*)
Add an application to a node

Parameters

topology [Topology] The topology

node [any hashable type] The ID of the node

name [str] The name of the application

attr_dict [dict, optional] Attributes of the application

****attr** [keyworded attributes] Attributes of the application

fnss.netconfig.nodeconfig.add_stack

add_stack (*topology*, *node*, *name*, *properties=None*, ***kwargs*)
Set stack on a node.

If the node already has a stack, it is overwritten

Parameters

topology [Topology] The topology

node [any hashable type] The ID of the node

name [str] The name of the stack

properties [dict, optional] The properties of the stack

****attr** [keyworded attributes] Further properties of the application

fnss.netconfig.nodeconfig.clear_applications

clear_applications (*topology*)
Remove all applications from all nodes of the topology

Parameters

topology [Topology] The topology

fnss.netconfig.nodeconfig.clear_stacks

clear_stacks (*topology*)
Remove all stacks from all nodes of the topology

Parameters

topology [Topology]

fnss.netconfig.nodeconfig.get_application_names

`get_application_names` (*topology*, *node*)

Return a list of names of applications deployed on a node

Parameters

topology [Topology] The topology

node [any hashable type] The ID of the node

Returns

application_names [list] A list of application names

fnss.netconfig.nodeconfig.get_application_properties

`get_application_properties` (*topology*, *node*, *name*)

Return a dictionary containing all the properties of an application deployed on a node

Parameters

topology [Topology] The topology

node [any hashable type] The ID of the node

name [str] The name of the application

Returns

applications [dict] A dictionary containing the properties of the application

fnss.netconfig.nodeconfig.get_stack

`get_stack` (*topology*, *node*, *data=True*)

Return the stack of a node, if any

Parameters

topology [Topology] The topology

node [any hashable type] The ID of the node

data [bool, optional] If true, returns a tuple of the stack name and its attributes, otherwise just the stack name

Returns

stack [tuple (name, properties) or name only] If data = True, a tuple of two values, where the first value is the name of the stack and the second value is the dictionary of its properties. If data = False returns only the stack name If no stack is deployed, return None

fnss.netconfig.nodeconfig.remove_application

`remove_application` (*topology*, *node*, *name=None*)

Remove an application from a node

Parameters

topology [Topology] The topology object

node [any hashable type] The ID of the node from which the application is to be removed

name [optional] The name of the application to remove. If not given, all the applications of the node are removed

fnss.netconfig.nodeconfig.remove_stack

remove_stack(*topology*, *node*)

Remove stack from a node

Parameters

topology [Topology] The topology

node [any hashable type] The ID of the node

`weights` module

Functions to assign and manipulate link weights to a network topology.

`clear_weights`(topology)	Remove all weights from the topology.
`get_weights`(topology)	Returns all the weights.
`set_weights_constant`(topology[, weight, links])	Assign a constant weight to all selected links
`set_weights_delays`(topology)	Assign link weights to links proportionally their delay.
`set_weights_inverse_capacity`(topology)	Assign link weights to links proportionally to the inverse of their capacity.

fnss.netconfig.weights.clear_weights

clear_weights(*topology*)

Remove all weights from the topology.

Parameters

topology [Topology]

fnss.netconfig.weights.get_weights

get_weights(*topology*)

Returns all the weights.

Parameters

topology [Topology]

Returns

weights [dict] Dictionary of weights keyed by link.

Examples

```
>>> import fnss
>>> topology = fnss.Topology()
>>> topology.add_path([1, 2, 3])
>>> fnss.set_weights_constant(topology, weight=2.0)
>>> weight = fnss.get_weights(topology)
>>> weight[(1,2)]
2.0
```

fnss.netconfig.weights.set_weights_constant

set_weights_constant (*topology*, *weight=1.0*, *links=None*)
Assign a constant weight to all selected links

Parameters

topology [Topology] The topology on which weights are applied.

weight [float, optional] The constant weight to be applied to all links

links [iterable, optional] Iterable container of selected links on which weights are applied. If it is None, all links are selected

Examples

```
>>> import fnss
>>> topology = fnss.Topology()
>>> topology.add_edges_from([(1, 2), (5, 8), (4, 5), (1, 7)])
>>> fnss.set_weights_constant(topology, weight=1.0, links=[(1, 2), (5, 8), (4,
↪5)])
```

fnss.netconfig.weights.set_weights_delays

set_weights_delays (*topology*)
Assign link weights to links proportionally their delay. Weights are normalized so that the minimum weight is 1.

Parameters

topology [Topology] The topology on which weights are applied.

Examples

```
>>> import fnss
>>> topology = fnss.erdos_renyi_topology(50, 0.1)
>>> fnss.set_delays_constant(topology, 2, 'ms')
>>> fnss.set_weights_delays(topology)
```

fnss.netconfig.weights.set_weights_inverse_capacity

set_weights_inverse_capacity(*topology*)

Assign link weights to links proportionally to the inverse of their capacity. Weights are normalized so that the minimum weight is 1.

Parameters

topology [Topology] The topology on which weights are applied.

Examples

```
>>> import fnss
>>> topology = fnss.Topology()
>>> topology.add_path([1,2,3,4])
>>> fnss.set_capacities_constant(topology, 10, 'Mbps')
>>> fnss.set_weights_inverse_capacity(topology)
```

1.3.2.2 traffic package

eventscheduling module

Functions and classes for creating and manipulating event schedules.

An event schedule is simply a list of events each labelled with a time and a number of properties.

An event schedule can be read and written from/to an XML files with provided functions.

deterministic_process_event_schedule(…)	Return a schedule of events separated by a fixed time interval
poisson_process_event_schedule(avg_interval, …)	Return a schedule of Poisson-distributed events
read_event_schedule(path)	Read event schedule from an XML file
write_event_schedule(event_schedule, path[, …])	Write an event schedule object to an XML file.

fnss.traffic.eventscheduling.deterministic_process_event_schedule

deterministic_process_event_schedule(*interval*, *t_start*, *duration*, *t_unit*, *event_generator*, **args*, ***kwargs*)

Return a schedule of events separated by a fixed time interval

Parameters

interval [float] The fixed time interval between subsequent events

t_start [float] The time at which the schedule starts

duration [float] The duration of the event schedule

t_unit: string The unit in which time values are expressed (e.g. ‘ms’, ‘s’)

event_generator [function] A function that when called returns an event, i.e. a dictionary of event properties

***args** [argument list] List of non-keyworded arguments for event_generator function

****kwargs** [keyworded argument list] List of keyworded arguments for event_generator function

Returns

event_schedule [EventSchedule] An EventSchedule object

fnss.traffic.eventscheduling.poisson_process_event_schedule

`poisson_process_event_schedule`(*avg_interval*, *t_start*, *duration*, *t_unit*, *event_generator*, **args*, ***kwargs*)

Return a schedule of Poisson-distributed events

Parameters

avg_interval [float] The average time interval between subsequent events

t_start [float] The time at which the schedule starts

duration [float] The duration of the event schedule

t_unit [string] The unit in which time values are expressed (e.g. 'ms', 's')

seed [int, long or hashable type, optional] The seed to be used by the random generator.

event_generator [callable] A function that when called returns an event, i.e. a dictionary of event properties

***args** [argument list] List of non-keyworded arguments for event_generator function

****kwargs** [keyworded argument list] List of keyworded arguments for event_generator function

Returns

event_schedule [EventSchedule] An EventSchedule object

fnss.traffic.eventscheduling.read_event_schedule

`read_event_schedule`(*path*)

Read event schedule from an XML file

Parameters

path [str] The path to the event schedule XML file

Returns

event_schedule [EventSchedule] The parsed event schedule

fnss.traffic.eventscheduling.write_event_schedule

`write_event_schedule`(*event_schedule*, *path*, *encoding='utf-8'*, *prettyprint=True*)

Write an event schedule object to an XML file.

Parameters

event_schedule [EventSchedule] The event schedule to write

path [str] The path of the output XML file

encoding [str, optional] The desired encoding of the output file

prettyprint [bool, optional] Specify whether the XML file should be written with indentation for improved human readability

`trafficmatrices` module

Functions and classes for creating and manipulating traffic matrices.

The functions of this class allow users to synthetically generate traffic matrices with given statistical properties according to models proposed in literature.

The output of this generation is either a TrafficMatrix or a TrafficMatrixSequence object.

A traffic matrix or a sequence of matrices can be read and written from/to an XML files with provided functions.

`link_loads`(topology, traffic_matrix[, . . .])	Calculate link utilization given a traffic matrix.
`read_traffic_matrix`(path[, encoding])	Parses a traffic matrix from a traffic matrix XML file.
`sin_cyclostationary_traffic_matrix`(topology, . . .)	Return a cyclostationary sequence of traffic matrices, where traffic volumes evolve over time as sin waves.
`static_traffic_matrix`(topology, mean, stddev)	Return a TrafficMatrix object, i.e.
`stationary_traffic_matrix`(topology, mean, . . .)	Return a stationary sequence of traffic matrices.
`validate_traffic_matrix`(topology, traffic_matrix)	Validate whether a given traffic matrix and given topology are compatible.
`write_traffic_matrix`(traffic_matrix, path[, . . .])	Write a TrafficMatrix or a TrafficMatrixSequence object to an XML file.

fnss.traffic.trafficmatrices.link_loads

`link_loads` (*topology*, *traffic_matrix*, *routing_matrix=None*, *ecmp=False*)

Calculate link utilization given a traffic matrix.

Return a dictionary mapping for each link of a topology, the relative link utilization (i.e. traffic volume divided by link capacity) given a traffic matrix. The keys of the dictionary are (u, v) tuple where u and v are respectively the source and destination nodes of the link. The values are float values between 0 and 1. A zero value means that the link is not utilized, while a one value means that the link is saturated.

Link utilizations are calculated assuming that all traffic is routed following the shortest path from origin to destination, calculated with the Dijkstra algorithm. If the topology is annotated with link weights, they are used for the shortest path calculation. Otherwise hop count is used.

Parameters

topology [topology] The topology whose link utilization is calculated. This topology must be annotate with at least link capacity. If it also presents link weights, those are used for shortest paths calculation.

tm [TrafficMatrix] The traffic matrix associated to the topology.

routing_matrix [dict of dicts] The routing matrix used by the traffic. This matrix is a dictionary of dictionaries, where the keys of the root dictionary are the origin nodes, the keys of the nested dictionary are the destination nodes and the values of the nested dictionary are lists of nodes on the path from origin to destination (both included). For example, if the path from node 1 to node 4 is 1 -> 2 -> 3 -> 4, then routing_matrix[1][4] = [1, 2, 3, 4]. If ecmp is set to True, the values of the nested dictionary are lists of lists of nodes, each representing a path, among which the load will be equally divided. The networkx all_pairs_dijkstra_path function returns shortest paths in this format. If this parameter is None, then Dijkstra shortest paths are used.

ecmp: bool Enables the usage of Equal-Cost Multi Path Routing.

Returns

link_loads [dict] A dictionary of link loads keyed by link

fnss.traffic.trafficmatrices.read_traffic_matrix

read_traffic_matrix(*path*, *encoding='utf-8'*)

Parses a traffic matrix from a traffic matrix XML file. If the XML file contains more than one traffic matrix, it returns a TrafficMatrixSequence object, otherwise a TrafficMatrixObject.

Parameters

path: str The path of the XML file to parse

encoding [str, optional] The encoding of the file

Returns

tm [TrafficMatrix or TrafficMatrixSequence]

fnss.traffic.trafficmatrices.sin_cyclostationary_traffic_matrix

sin_cyclostationary_traffic_matrix(*topology*, *mean*, *stddev*, *gamma*, *log_psi*, *delta=0.2*, *n=24*, *periods=1*, *max_u=0.9*, *origin_nodes=None*, *destination_nodes=None*)

Return a cyclostationary sequence of traffic matrices, where traffic volumes evolve over time as sin waves.

The sequence is generated by first generating a single matrix assigning traffic volumes drawn from a lognormal distribution and assigned to specific origin-destination pairs using the Ranking Metrics Heuristic method proposed by Nucci et al. *[3]*. Then, all matrices of the sequence are generated by adding zero-mean normal fluctuation in the traffic volumes. Finally, traffic volumes are multiplied by a sin function with unitary mean to model periodic fluctuations.

This process was originally proposed by *[3]*.

Cyclostationary sequences of traffic matrices are generally suitable for modeling real network traffic over long periods, up to several days. In fact, real traffic exhibits diurnal patterns well modelled by cyclostationary sequences.

Parameters

topology [topology] The topology for which the traffic matrix is calculated. This topology can either be directed or undirected. If it is undirected, this function assumes that all links are full-duplex.

mean [float] The mean volume of traffic among all origin-destination pairs

stddev [float] The standard deviation of volumes among all origin-destination pairs.

gamma [float] Parameter expressing relation between mean and standard deviation of traffic volumes of a specific flow over the time

log_psi [float] Parameter expressing relation between mean and standard deviation of traffic volumes of a specific flow over the time

delta [float [0, 1]] A parameter indicating the intensity of variation of traffic volumes over a period. Specifically, let x be the mean volume over a specific OD pair, the minimum and maximum traffic volumes for that OD pair (excluding random fluctuations) are respectively $x * (1 - delta)$ and $x * (1 + delta)$

n [int] Number of traffic matrices per period. For example, if it is desired to model traffic varying cyclically over a 24 hour period, and n is set to 24, therefore, the time interval between subsequent traffic matrices is is 1 hour.

periods [int] Number of periods. In total the sequence is composed of $n * periods$ traffic matrices.

max_u [float, optional] Represent the max link utilization. If specified, traffic volumes are scaled so that the most utilized link of the network has an utilization equal to max_u. If None, volumes are not scaled, but in this case links may end up with an utilization factor greater than 1.0

origin_nodes [list, optional] A list of all nodes which can be traffic sources. If not specified all nodes of the topology are traffic sources

destination_nodes [list, optional] A list of all nodes which can be traffic destinations. If not specified all nodes of the topology are traffic destinations

Returns

tms [TrafficMatrixSequence]

References

[3]

fnss.traffic.trafficmatrices.static_traffic_matrix

static_traffic_matrix(*topology*, *mean*, *stddev*, *max_u=0.9*, *origin_nodes=None*, *destination_nodes=None*)

Return a TrafficMatrix object, i.e. a single traffic matrix, representing the traffic volume exchanged over a network at a specific point in time

This matrix is generated by assigning traffic volumes drawn from a lognormal distribution and assigned to specific origin-destination pairs using the Ranking Metrics Heuristic method proposed by Nucci et al. *[1]*

Parameters

topology [topology] The topology for which the traffic matrix is calculated. This topology can either be directed or undirected. If it is undirected, this function assumes that all links are full-duplex.

mean [float] The mean volume of traffic among all origin-destination pairs

stddev [float] The standard deviation of volumes among all origin-destination pairs.

max_u [float, optional] Represent the max link utilization. If specified, traffic volumes are scaled so that the most utilized link of the network has an utilization equal to max_u. If None, volumes are not scaled, but in this case links may end up with an utilization factor greater than 1.0

origin_nodes [list, optional] A list of all nodes which can be traffic sources. If not specified, all nodes of the topology are traffic sources

destination_nodes [list, optional] A list of all nodes which can be traffic destinations. If not specified, all nodes of the topology are traffic destinations

Returns

tm [TrafficMatrix]

References

[1]

fnss.traffic.trafficmatrices.stationary_traffic_matrix

`stationary_traffic_matrix`(*topology*, *mean*, *stddev*, *gamma*, *log_psi*, *n*, *max_u=0.9*, *origin_nodes=None*, *destination_nodes=None*)

Return a stationary sequence of traffic matrices.

The sequence is generated by first generating a single matrix assigning traffic volumes drawn from a lognormal distribution and assigned to specific origin-destination pairs using the Ranking Metrics Heuristic method proposed by Nucci et al. *[2]*. Then, all matrices of the sequence are generated by adding zero-mean normal fluctuation in the traffic volumes. This process was originally proposed by *[2]*

Stationary sequences of traffic matrices are generally suitable for modeling network traffic over short periods (up to 1.5 hours). Over longer periods, real traffic exhibits diurnal patterns and they are better modelled by cyclostationary sequences

Parameters

topology [topology] The topology for which the traffic matrix is calculated. This topology can either be directed or undirected. If it is undirected, this function assumes that all links are full-duplex.

mean [float] The mean volume of traffic among all origin-destination pairs

stddev [float] The standard deviation of volumes among all origin-destination pairs.

gamma [float] Parameter expressing relation between mean and standard deviation of traffic volumes of a specific flow over the time

log_psi [float] Parameter expressing relation between mean and standard deviation of traffic volumes of a specific flow over the time

n [int] Number of matrices in the sequence

max_u [float, optional] Represent the max link utilization. If specified, traffic volumes are scaled so that the most utilized link of the network has an utilization equal to max_u. If None, volumes are not scaled, but in this case links may end up with an utilization factor greater than 1.0

origin_nodes [list, optional] A list of all nodes which can be traffic sources. If not specified all nodes of the topology are traffic sources

destination_nodes [list, optional] A list of all nodes which can be traffic destinations. If not specified all nodes of the topology are traffic destinations

Returns

tms [TrafficMatrixSequence]

References

[2]

fnss.traffic.trafficmatrices.validate_traffic_matrix

validate_traffic_matrix(*topology*, *traffic_matrix*, *validate_load=False*)

Validate whether a given traffic matrix and given topology are compatible.

Returns True if they are compatible, False otherwise

This validation includes validating whether the origin-destination pairs of the traffic matrix are coincide with or are a subset of the origin-destination pairs of the topology. Optionally, this function can verify if the volumes of the traffic matrix are compatible too, i.e. if at any time, no link has an utilization greater than 1.0.

Parameters

topology [topology] The topology agains which the traffic matrix is validated

tm [TrafficMatrix or TrafficMatrixSequence] The traffic matrix (or sequence of) to be validated

validate_load [bool, optional] Specify whether load compatibility has to be validated or not. Default value is False

Returns

is_valid [bool] True if the topology and the traffic matrix are compatible, False otherwise

fnss.traffic.trafficmatrices.write_traffic_matrix

write_traffic_matrix(*traffic_matrix*, *path*, *encoding='utf-8'*, *prettyprint=True*)

Write a TrafficMatrix or a TrafficMatrixSequence object to an XML file. This function can be use to either persistently store a traffic matrix for later use or to export it to an FNSS adapter for a simulator or an API for another programming language.

Parameters

traffic_matrix [TrafficMatrix or TrafficMatrixSequence] The traffic matrix to save

path [str] The path where the file will be saved

encoding [str, optional] The desired encoding of the output file

prettyprint [bool, optional] Specify whether the XML file should be written with indentation for improved human readability

1.3.2.3 `topologies` package

`datacenter` module

Functions to generate commonly adopted datacenter topologies.

Each topology generation function returns an instance of DatacenterTopology

bcube_topology(n, k)	Return a Bcube datacenter topology, as described in *[R48460de4c968-1]*:

Continued on next page

Table 20 – continued from previous page

`fat_tree_topology`(k)	Return a fat tree datacenter topology, as described in *[Rdaad0f90b4be-1]*
`three_tier_topology`(n_core, n_aggregation, ...)	Return a three-tier data center topology.
`two_tier_topology`(n_core, n_edge, n_hosts)	Return a two-tier datacenter topology.

fnss.topologies.datacenter.bcube_topology

bcube_topology(*n*, *k*)

Return a Bcube datacenter topology, as described in *[1]*:

The BCube topology is a topology specifically designed for shipping-container based, modular data centers. A BCube topology comprises hosts with multiple network interfaces connected to commodity switches. It has the peculiar characteristic that switches are never directly connected to each other and hosts are used also for packet forwarding. This topology is defined as a recursive structure. A $Bcube_0$ is composed of n hosts connected to an n-port switch. A $Bcube_1$ is composed of n $Bcube_0$ connected to n n-port switches. A $Bcube_k$ is composed of n $Bcube_{k-1}$ connected to n^k n-port switches.

This topology comprises:

- $n^(k+1)$ hosts, each of them connected to $k+1$ switches
- $n*(k+1)$ switches, each of them having n ports

Each node has an attribute type which can either be *switch* or *host* and an attribute *level* which specifies at what level of the Bcube hierarchy it is located.

Each edge also has the attribute *level*.

Parameters

k [int] The level of Bcube

n [int] The number of host per $Bcube_0$

Returns

topology [DatacenterTopology]

References

[1]

fnss.topologies.datacenter.fat_tree_topology

fat_tree_topology(*k*)

Return a fat tree datacenter topology, as described in *[1]*

A fat tree topology built using k-port switches can support up to $(k^3)/4$ hosts. This topology comprises k pods with two layers of $k/2$ switches each. In each pod, each aggregation switch is connected to all the $k/2$ edge switches and each edge switch is connected to $k/2$ hosts. There are $(k/2)^2$ core switches, each of them connected to one aggregation switch per pod.

Each node has three attributes:

- type: can either be *switch* or *host*

- tier: can either be *core*, *aggregation*, *edge* or *leaf*. Nodes in
- pod: the pod id in which the node is located, unless it is a core switch the leaf tier are only host, while all core, aggregation and edge nodes are switches.

Each edge has an attribute type as well which can either be *core_edge* if it connects a core and an aggregation switch, *aggregation_edge*, if it connects an aggregation and a core switch or *edge_leaf* if it connects an edge switch to a host.

Parameters

k [int] The number of ports of the switches

Returns

topology [DatacenterTopology]

References

[1]

fnss.topologies.datacenter.three_tier_topology

`three_tier_topology`(*n_core*, *n_aggregation*, *n_edge*, *n_hosts*)

Return a three-tier data center topology.

This topology comprises switches organized in three tiers (core, aggregation and edge) and hosts connected to edge routers. Each core switch is connected to each aggregation, each edge switch is connected to one aggregation switch and finally each host is connected to exactly one edge switch.

Each node has two attributes:

- type: can either be *switch* or *host*
- tier: can either be *core*, *aggregation*, *edge* or *leaf*. Nodes in the leaf tier are only host, while all core, aggregation and edge nodes are switches.

Each edge has an attribute type as well which can either be *core_edge* if it connects a core and an aggregation switch, *aggregation_edge*, if it connects an aggregation and a core switch or *edge_leaf* if it connects an edge switch to a host.

The total number of hosts is $n_{aggregation} * n_{edge} * n_{hosts}$.

Parameters

n_core [int] Total number of core switches

n_aggregation [int] Total number of aggregation switches

n_edge [int] Number of edge switches per each each aggregation switch

n_hosts [int] Number of hosts connected to each edge switch.

Returns

topology [DatacenterTopology]

fnss.topologies.datacenter.two_tier_topology

two_tier_topology(*n_core*, *n_edge*, *n_hosts*)

Return a two-tier datacenter topology.

This topology comprises switches organized in two tiers (core and edge) and hosts connected to edge routers. Each core switch is connected to each edge switch while each host is connected to exactly one edge switch.

Each node has two attributes:

- type: can either be *switch* or *host*
- tier: can either be *core*, *edge* or *leaf*. Nodes in the leaf tier are only host, while all core and edge nodes are switches.

Each edge has an attribute type as well which can either be *core_edge* if it connects a core and an edge switch or *edge_leaf* if it connects an edge switch to a host.

Parameters

n_core [int] Total number of core switches

n_edge [int] Total number of edge switches

n_hosts [int] Number of hosts connected to each edge switch.

Returns

topology [DatacenterTopology]

`parsers` module

Functions to parse topologies from datasets or from other generators.

`parse_abilene`(topology_path[, links_path])	Parse the Abilene topology.
`parse_ashiip`(path)	Parse a topology from an output file generated by the aShiip topology generator
`parse_brite`(path[, capacity_unit, …])	Parse a topology from an output file generated by the BRITE topology generator
`parse_caida_as_relationships`(path)	Parse a topology from the CAIDA AS relationships dataset
`parse_inet`(path)	Parse a topology from an output file generated by the Inet topology generator
`parse_rocketfuel_isp_map`(path)	Parse a network topology from RocketFuel ISP map file.
`parse_rocketfuel_isp_latency`(latencies_path)	Parse a network topology from RocketFuel ISP topology file (latency.intra) with inferred link latencies and optionally annotate the topology with inferred weights (weights.infra).
`parse_topology_zoo`(path)	Parse a topology from the Topology Zoo dataset.

fnss.topologies.parsers.parse_abilene

parse_abilene(*topology_path*, *links_path=None*)

Parse the Abilene topology.

Parameters

topology_path [str] The path of the Abilene topology file

links_path [str, optional] The path of the Abilene links file

Returns

topology [DirectedTopology]

fnss.topologies.parsers.parse_ashiip

parse_ashiip(*path*)

Parse a topology from an output file generated by the aShiip topology generator

Parameters

path [str] The path to the aShiip output file

Returns

topology [Topology]

fnss.topologies.parsers.parse_brite

parse_brite(*path*, *capacity_unit='Mbps'*, *delay_unit='ms'*, *distance_unit='Km'*, *directed=True*)

Parse a topology from an output file generated by the BRITE topology generator

Parameters

path [str] The path to the BRITE output file

capacity_unit [str, optional] The unit in which link capacity values are expresses in the BRITE file

delay_unit [str, optional] The unit in which link delay values are expresses in the BRITE file

distance_unit [str, optional] The unit in which node coordinates are expresses in the BRITE file

directed [bool, optional] If True, the topology is parsed as directed topology.

Returns

topology [Topology or DirectedTopology]

Notes

Each node of the returned topology object is labeled with *latitude* and *longitude* attributes. These attributes are not expressed in degrees but in *distance_unit*.

fnss.topologies.parsers.parse_caida_as_relationships

parse_caida_as_relationships(*path*)

Parse a topology from the CAIDA AS relationships dataset

Parameters

path [str] The path to the CAIDA AS relationships file

Returns

topology [DirectedTopology]

Notes

The node names of the returned topology are the the ASN of the of the AS they represent and edges are annotated with the relationship between ASes they connect. The relationship values can either be *customer*, *peer* or *sibling*.

References

http://www.caida.org/data/active/as-relationships/ http://as-rank.caida.org/data/

fnss.topologies.parsers.parse_inet

parse_inet (*path*)

Parse a topology from an output file generated by the Inet topology generator

Parameters

path [str] The path to the Inet output file

Returns

topology [Topology]

Notes

Each node of the returned topology object is labeled with *latitude* and *longitude* attributes. These attributes are not expressed in degrees but in Kilometers.

fnss.topologies.parsers.parse_rocketfuel_isp_map

parse_rocketfuel_isp_map (*path*)

Parse a network topology from RocketFuel ISP map file.

The ASes provided by the RocketFuel dataset are the following:

ASN	Name	Span	Region	Nodes (r1)	Nodes (r0)
1221 1239 1755 2914 3257 3356 3967 4755 6461 7018	Telstra (Australia) Sprintlink (US) EBONE (Europe) Verio (US) Tiscali (Europe) Level 3 (US) Exodus (US) VSNL (India) Abovenet (US) AT&T (US)	world world world world world world world world world world	AUS US Europe US Europe US US India US US	2999 8352 609 **7109** 855 **3447** 917 121 0 10152	378 (318) 700 (604) 172 **1013** 248 (240) 652 215 (201) 12 202 656 (631)

Parameters

path [str] The path of the file containing the RocketFuel map. It should have extension .cch

Returns

topology [DirectedTopology] The object containing the parsed topology.

Raises

ValueError If the provided file cannot be parsed correctly.

Notes

The returned topology is always directed. If an undirected topology is desired, convert it using the DirectedTopology.to_undirected() method.

Each node of the returned graph has the following attributes:

- **type**: string
- **location**: string (optional)
- **address**: string
- **r**: int
- **backbone**: boolean (optional)

Each edge of the returned graph has the following attributes:

- type : string, which can either be *internal* or *external*

If the topology contains self-loops (links starting and ending in the same node) they are stripped from the topology.

Examples

```
>>> import fnss
>>> topology = fnss.parse_rocketfuel_isp_map('1221.r0.cch')
```

fnss.topologies.parsers.parse_rocketfuel_isp_latency

parse_rocketfuel_isp_latency(*latencies_path*, *weights_path=None*)

Parse a network topology from RocketFuel ISP topology file (latency.intra) with inferred link latencies and optionally annotate the topology with inferred weights (weights.infra).

The ASes provided by the RocketFuel dataset are the following:

ASN	Name	Span	Region	Nodes	Lrgst conn. comp.
1221 1239 1755 3257 3967 6461	Telstra (Australia) Sprintlink (US) EBONE (Europe) Tiscali (Europe) Exodus (US) Abovenet (US)	world world world world world world	AUS US Europe Europe US US	108 315 87 **161** 79 141	104 315 87 **161** 79 138

Parameters

latencies_path [str] The path of the file containing the RocketFuel latencies file. It should have extension .intra

weights_path [str, optional] The path of the file containing the RocketFuel weights file. It should have extension .intra

Returns

topology [DirectedTopology] The object containing the parsed topology.

Notes

The returned topology is directed. It can be converted using the DirectedTopology.to_undirected() method if an undirected topology is desired.

Each node of the returned graph has the following attributes:

- **name**: string
- **location**: string

Each edge of the returned graph has the following attributes:

- **delay** : int
- **wdights** : float (only if a weights file was specified)

Examples

```
>>> import fnss
>>> topology = fnss.parse_rocketfuel_isp_latency('1221.latencies.intra')
```

fnss.topologies.parsers.parse_topology_zoo

parse_topology_zoo (*path*)
Parse a topology from the Topology Zoo dataset.

Parameters

path [str] The path to the Topology Zoo file

Returns

topology [Topology or DirectedTopology] The parsed topology.

Notes

If the parsed topology contains bundled links, i.e. multiple links between the same pair or nodes, the topology is parsed correctly but each bundle of links is represented as a single link whose capacity is the sum of the capacities of the links of the bundle (if capacity values were provided). The returned topology has a boolean attribute named *link_bundling* which is True if the topology contains at list one bundled link or False otherwise. If the topology contains bundled links, then each link has an additional boolean attribute named *bundle* which is True if that specific link was bundled in the original topology or False otherwise.

`randmodels` module

Functions to generate random topologies according to a number of models.

The generated topologies are either Topology or DirectedTopology objects.

`barabasi_albert_topology`(n, m, m0[, seed])	Return a random topology using Barabasi-Albert preferential attachment model.
`erdos_renyi_topology`(n, p[, seed, fast])	Return a random graph $G_{n,p}$ (Erdos-Renyi graph, binomial graph).
`extended_barabasi_albert_topology`(n, m, m0, p, q)	Return a random topology using the extended Barabasi-Albert preferential attachment model.
`glp_topology`(n, m, m0, p, beta[, seed])	Return a random topology using the Generalized Linear Preference (GLP) preferential attachment model.
`waxman_1_topology`(n[, alpha, beta, L, . . .])	Return a Waxman-1 random topology.
`waxman_2_topology`(n[, alpha, beta, domain, . . .])	Return a Waxman-2 random topology.

fnss.topologies.randmodels.barabasi_albert_topology

`barabasi_albert_topology`(*n*, *m*, *m0*, *seed=None*)

Return a random topology using Barabasi-Albert preferential attachment model.

A topology of n nodes is grown by attaching new nodes each with m links that are preferentially attached to existing nodes with high degree.

More precisely, the Barabasi-Albert topology is built as follows. First, a line topology with m0 nodes is created. Then at each step, one node is added and connected to m existing nodes. These nodes are selected randomly with probability

$$\Pi(i) = \frac{deg(i)}{sum_{v \in V} deg V}.$$

Where i is the selected node and V is the set of nodes of the graph.

Parameters

n [int] Number of nodes

m [int] Number of edges to attach from a new node to existing nodes

m0 [int] Number of nodes initially attached to the network

seed [int, optional] Seed for random number generator (default=None).

Returns

G [Topology]

Notes

The initialization is a graph with with m nodes connected by $m - 1$ edges. It does not use the Barabasi-Albert method provided by NetworkX because it does not allow to specify *m0* parameter. There are no disconnected subgraphs in the topology.

References

[1]

fnss.topologies.randmodels.erdos_renyi_topology

`erdos_renyi_topology` (*n*, *p*, *seed=None*, *fast=False*)
Return a random graph $G_{n,p}$ (Erdos-Renyi graph, binomial graph).

Chooses each of the possible edges with probability p.

Parameters

n [int] The number of nodes.

p [float] Probability for edge creation.

seed [int, optional] Seed for random number generator (default=None).

fast [boolean, optional] Uses the algorithm proposed by *[3]*, which is faster for small p

References

[1], *[2]*, *[3]*

fnss.topologies.randmodels.extended_barabasi_albert_topology

`extended_barabasi_albert_topology` (*n*, *m*, *m0*, *p*, *q*, *seed=None*)
Return a random topology using the extended Barabasi-Albert preferential attachment model.

Differently from the original Barabasi-Albert model, this model takes into account the presence of local events, such as the addition of new links or the rewiring of existing links.

More precisely, the Barabasi-Albert topology is built as follows. First, a topology with *m0* isolated nodes is created. Then, at each step: with probability *p* add *m* new links between existing nodes, selected with probability:

$$\Pi(i) = \frac{deg(i) + 1}{\sum_{v \in V}(deg(v) + 1)}$$

with probability *q* rewire *m* links. Each link to be rewired is selected as follows: a node i is randomly selected and a link is randomly removed from it. The node i is then connected to a new node randomly selected with probability $\Pi(i)$, with probability $1 - p - q$ add a new node and attach it to m nodes of the existing topology selected with probability $\Pi(i)$

Repeat the previous step until the topology comprises n nodes in total.

Parameters

n [int] Number of nodes

m [int] Number of edges to attach from a new node to existing nodes

m0 [int] Number of edges initially attached to the network

p [float] The probability that new links are added

q [float] The probability that existing links are rewired

seed [int, optional] Seed for random number generator (default=None).

Returns

G [Topology]

References

[1]

fnss.topologies.randmodels.glp_topology

glp_topology(*n*, *m*, *m0*, *p*, *beta*, *seed=None*)

Return a random topology using the Generalized Linear Preference (GLP) preferential attachment model.

It differs from the extended Barabasi-Albert model in that there is link rewiring and a beta parameter is introduced to fine-tune preferential attachment.

More precisely, the GLP topology is built as follows. First, a line topology with *m0* nodes is created. Then, at each step: with probability p, add m new links between existing nodes, selected with probability:

$$\Pi(i) = \frac{deg(i) - \beta 1}{\sum_{v \in V}(deg(v) - \beta)}$$

with probability $1 - p$, add a new node and attach it to m nodes of the existing topology selected with probability $\Pi(i)$

Repeat the previous step until the topology comprises n nodes in total.

Parameters

n [int] Number of nodes

m [int] Number of edges to attach from a new node to existing nodes

m0 [int] Number of edges initially attached to the network

p [float] The probability that new links are added

beta [float] Parameter to fine-tune preferntial attachment: beta < 1

seed [int, optional] Seed for random number generator (default=None).

Returns

G [Topology]

References

[1]

fnss.topologies.randmodels.waxman_1_topology

waxman_1_topology(*n*, *alpha=0.4*, *beta=0.1*, *L=1.0*, *distance_unit='Km'*, *seed=None*)

Return a Waxman-1 random topology.

The Waxman-1 random topology models assigns link between nodes with probability

$$p = \alpha * exp(-d/(\beta * L)).$$

where the distance *d* is chosen randomly in *[0,L]*.

Parameters

n [int] Number of nodes

alpha [float] Model parameter chosen in *(0,1]* (higher alpha increases link density)

beta [float] Model parameter chosen in *(0,1]* (higher beta increases difference between density of short and long links)

L [float] Maximum distance between nodes.

seed [int, optional] Seed for random number generator (default=None).

Returns

G [Topology]

Notes

Each node of G has the attributes *latitude* and *longitude*. These attributes are not expressed in degrees but in *distance_unit*.

Each edge of G has the attribute *length*, which is also expressed in *distance_unit*.

References

[1]

fnss.topologies.randmodels.waxman_2_topology

waxman_2_topology(*n*, *alpha=0.4*, *beta=0.1*, *domain=(0, 0, 1, 1)*, *distance_unit='Km'*, *seed=None*)

Return a Waxman-2 random topology.

The Waxman-2 random topology models place n nodes uniformly at random in a rectangular domain. Two nodes u, v are connected with a link with probability

$$p = \alpha * exp(-d/(\beta * L)).$$

where the distance *d* is the Euclidean distance between the nodes u and v. and *L* is the maximum distance between all nodes in the graph.

Parameters

n [int] Number of nodes

alpha [float] Model parameter chosen in *(0,1]* (higher alpha increases link density)

beta [float] Model parameter chosen in *(0,1]* (higher beta increases difference between density of short and long links)

domain [tuple of numbers, optional] Domain size (xmin, ymin, xmax, ymax)

seed [int, optional] Seed for random number generator (default=None).

Returns

G [Topology]

Notes

Each edge of G has the attribute *length*

References

[1]

`simplemodels` module

Generate canonical deterministic topologies

chord_topology(m[, r])	Return a Chord topology, as described in *[R409142fb3296-1]*:
dumbbell_topology(m1, m2)	Return a dumbbell topology consisting of two star topologies connected by a path.
full_mesh_topology(n)	Return a fully connected mesh topology of n nodes
k_ary_tree_topology(k, h)	Return a balanced k-ary tree topology of with depth h
line_topology(n)	Return a line topology of n nodes
ring_topology(n)	Return a ring topology of n nodes
star_topology(n)	Return a star (a.k.a hub-and-spoke) topology of $n + 1$ nodes

fnss.topologies.simplemodels.chord_topology

chord_topology (*m*, *r=1*)

Return a Chord topology, as described in *[1]*:

Chord is a Distributed Hash Table (DHT) providing guaranteed correctness. In Chord, both nodes and keys are identified by sequences of m bits. Keys can be resolved in at most $log(n)$ steps (with n being the number of nodes) as long as each node maintains a routing table o n entries.

In this implementation, it is possible only to generate topologies with a number of nodes $n = 2^m$. where m is the length (in bits) of the keys used by Chord and also corresponds the the size of the finger table kept by each node.

The r argument is the number of nearest successors which can be optionally kept at each node to guarantee correctness in case of node failures.

Parameters

m [int] The length of keys (in bits), which also corresponds to the length of the finger table of each node

r [int, optional] The length of the nearest successors table

Returns

G [DirectedTopology] A Chord topology

References

[1]

fnss.topologies.simplemodels.dumbbell_topology

dumbbell_topology(*m1*, *m2*)

Return a dumbbell topology consisting of two star topologies connected by a path.

More precisely, two star graphs K_{m1} form the left and right bells, and are connected by a path P_{m2}.

The $2 * m1 + m2$ nodes are numbered as follows.

- $0, ..., m1 - 1$ for the left barbell,
- $m1, ..., m1 + m2 - 1$ for the path,
- $m1 + m2, ..., 2 * m1 + m2 - 1$ for the right barbell.

The 3 subgraphs are joined via the edges $(m1 - 1, m1)$ and $(m1 + m2 - 1, m1 + m2)$. If m2 = 0, this is merely two star topologies joined together.

Please notice that this dumbbell topology is different from the barbell graph generated by networkx's barbell_graph function. That barbell graph consists of two complete graphs connected by a path. This consists of two stars whose roots are connected by a path. This dumbbell topology is particularly useful for simulating transport layer protocols.

All nodes and edges of this topology have an attribute *type* which can be either *right bell*, *core* or *left_bell*

Parameters

m1 [int] The number of nodes in each bell

m2 [int] The number of nodes in the path

Returns

topology [A Topology object]

fnss.topologies.simplemodels.full_mesh_topology

full_mesh_topology(*n*)

Return a fully connected mesh topology of n nodes

Parameters

n [int] The number of nodes

Returns

topology [A Topology object]

fnss.topologies.simplemodels.k_ary_tree_topology

k_ary_tree_topology(*k*, *h*)

Return a balanced k-ary tree topology of with depth h

Each node has two attributes:

- type: which can either be *root*, *intermediate* or *leaf*
- depth:math:*(0, h)* the height of the node in the tree, where 0 is the root and h are leaves.

Parameters

k [int] The branching factor of the tree

h [int] The height or depth of the tree

Returns

topology [A Topology object]

fnss.topologies.simplemodels.line_topology

line_topology(*n*)

Return a line topology of n nodes

Parameters

n [int] The number of nodes

Returns

topology [A Topology object]

fnss.topologies.simplemodels.ring_topology

ring_topology(*n*)

Return a ring topology of n nodes

Parameters

n [int] The number of nodes

Returns

topology [A Topology object]

fnss.topologies.simplemodels.star_topology

star_topology(*n*)

Return a star (a.k.a hub-and-spoke) topology of $n + 1$ nodes

The root (hub) node has id 0 while all leaf (spoke) nodes have id $(1, n + 1)$.

Each node has the attribute type which can either be *root* (for node 0) or *leaf* for all other nodes

Parameters

n [int] The number of leaf nodes

Returns

topology [A Topology object]

`topology` module

Basic functions and classes for operating on network topologies.

fan_in_out_capacities(topology)	Calculate fan-in and fan-out capacities for all nodes of the topology.
od_pairs_from_topology(topology)	Calculate all possible origin-destination pairs of the topology.
rename_edge_attribute(topology, old_attr, ...)	Rename all edges attributes with a specific name to a new name
rename_node_attribute(topology, old_attr, ...)	Rename all nodes attributes with a specific name to a new name
read_topology(path[, encoding])	Read a topology from an XML file and returns either a Topology or a DirectedTopology object
write_topology(topology, path[, encoding, ...])	Write a topology object on an XML file

fnss.topologies.topology.fan_in_out_capacities

fan_in_out_capacities(*topology*)

Calculate fan-in and fan-out capacities for all nodes of the topology.

The fan-in capacity of a node is the sum of capacities of all incoming links, while the fan-out capacity is the sum of capacities of all outgoing links.

Parameters

topology [Topology] The topology object whose fan-in and fan-out capacities are calculated. This topology must be annotated with link capacities.

Returns

fan_in_out_capacities [tuple (fan_in, fan_out)] A tuple of two dictionaries, representing, respectively the fan-in and fan-out capacities keyed by node.

Notes

This function works correctly for both directed and undirected topologies. If the topology is undirected, the returned dictionaries of fan-in and fan-out capacities are identical.

Examples

```
>>> import fnss
>>> topology = fnss.star_topology(3)
>>> fnss.set_capacities_constant(topology, 10, 'Mbps')
>>> in_cap, out_cap = fnss.fan_in_out_capacities(topology)
>>> in_cap
{0: 30, 1: 10, 2: 10, 3: 10}
>>> out_cap
{0: 30, 1: 10, 2: 10, 3: 10}
```

fnss.topologies.topology.od_pairs_from_topology

od_pairs_from_topology(*topology*)

Calculate all possible origin-destination pairs of the topology. This function does not simply calculate all possible pairs of the topology nodes. Instead, it only returns pairs of nodes connected by at least a path.

Parameters

topology [Topology or DirectedTopology] The topology whose OD pairs are calculated

Returns

od_pair [list] List containing all origin destination tuples.

Examples

```
>>> import fnss
>>> topology = fnss.ring_topology(3)
>>> fnss.od_pairs_from_topology(topology)
[(0, 1), (0, 2), (1, 0), (1, 2), (2, 0), (2, 1)]
```

fnss.topologies.topology.rename_edge_attribute

rename_edge_attribute(*topology*, *old_attr*, *new_attr*)

Rename all edges attributes with a specific name to a new name

Parameters

topology [Topology] The topology object

old_attr [any hashable type] Old attribute name

new_attr [any hashable type] New attribute name

fnss.topologies.topology.rename_node_attribute

rename_node_attribute(*topology*, *old_attr*, *new_attr*)

Rename all nodes attributes with a specific name to a new name

Parameters

topology [Topology] The topology object

old_attr [any hashable type] Old attribute name

new_attr [any hashable type] New attribute name

fnss.topologies.topology.read_topology

read_topology(*path*, *encoding='utf-8'*)

Read a topology from an XML file and returns either a Topology or a DirectedTopology object

Parameters

path [str] The path of the topology XML file to parse

encoding [str, optional] The encoding of the file

Returns

topology: Topology or DirectedTopology The parsed topology

fnss.topologies.topology.write_topology

write_topology (*topology*, *path*, *encoding='utf-8'*, *prettyprint=True*)
Write a topology object on an XML file

Parameters

topology [Topology] The topology object to write

path [str] The file ob which the topology will be written

encoding [str, optional] The encoding of the target file

prettyprint [bool, optional] Indent the XML code in the output file

1.3.2.4 adapters package

autonetkit module

Adapter for AutoNetkit.

This module contains function for converting FNSS Topology objects into NetworkX graph objects compatible with AutoNetKit and viceversa.

from_autonetkit(topology)	Convert an AutoNetKit graph into an FNSS Topology object.
to_autonetkit(topology)	Convert an FNSS topology into a NetworkX graph object compatible for AutoNetKit.

fnss.adapters.autonetkit.from_autonetkit

from_autonetkit (*topology*)
Convert an AutoNetKit graph into an FNSS Topology object.

The current implementation of this function only renames the weight attribute from *weight* to *ospf_cost*

Parameters

topology [NetworkX graph] An AutoNetKit NetworkX graph

Returns

fnss_topology [FNSS Topology] FNSS topology

fnss.adapters.autonetkit.to_autonetkit

to_autonetkit (*topology*)
Convert an FNSS topology into a NetworkX graph object compatible for AutoNetKit.

The returned graph can be saved into a GraphML file using NetworkX *write_graphml* function and then passed to AutoNetKit as command line parameter.

The current implementation of this function only renames the weight attribute from *weight* to *ospf_cost*

Parameters

topology [FNSS Topology] Autonetkit topology object

Returns

ank_graph [FNSS topology] an FNSS topology compatible for import to AutoNetKit

`jfed` module

Adapter for jFed

Provides function to convert an FNSS Topology object into a jFed rspec file and viceversa.

jFed <http://jfed.iminds.be/>_ is a Java-based framework to support the integration of federated testbed, developed by *iMinds <http://www.iminds.be/>_* in the contex of the *Fed4FIRE <http://www.fed4fire.eu/>_* project funded by the Framework Programme 7 (FP7) of the European Union.

`from_jfed`(path)	Read a jFed RSPEC file and returns an FNSS topology modelling the network topology of the jFed experiment specification.
`to_jfed`(topology, path[, testbed, encoding, . . .])	Convert a topology object into an RSPEC file for jFed

fnss.adapters.jfed.from_jfed

`from_jfed`(*path*)

Read a jFed RSPEC file and returns an FNSS topology modelling the network topology of the jFed experiment specification.

Parameters

path [str] The path of the jFed RSPEC file to parse

Returns

topology: Topology The parsed topology

Notes

This function does not support directed topologies and unidirectional links

It is possible in jFed to create multipoint links (links with more than 2 endpoints). Such types of link cannot be modelled in FNSS. Therefore, any attempt to convert an RSPEC with such links will fail.

fnss.adapters.jfed.to_jfed

`to_jfed`(*topology*, *path*, *testbed='wall1.ilabt.iminds.be'*, *encoding='utf-8'*, *prettyprint=True*)

Convert a topology object into an RSPEC file for jFed

Parameters

topology [Topology] The topology object

path [str] The file to which the RSPEC will be written

testbed [str, optional] URI of the testbed to use

encoding [str, optional] The encoding of the target file

prettyprint [bool, optional] Indent the XML code in the output file

Notes

It currently supports only undirected topologies, if a topology is directed it is converted to undirected

`mn` module

Adapter for Mininet.

This module contains function to convert FNSS topologies into Mininet topologies and viceversa.

from_mininet(topology)	Convert a Mininet topology to an FNSS one.
to_mininet(topology[, switches, hosts, …])	Convert an FNSS topology to Mininet Topo object that can be used to deploy a Mininet network.

fnss.adapters.mn.from_mininet

from_mininet (*topology*)

Convert a Mininet topology to an FNSS one.

Parameters

topology [Mininet Topo] A Mininet topology object

Returns

topology [Topology] An FNSS Topology object

fnss.adapters.mn.to_mininet

to_mininet (*topology*, *switches=None*, *hosts=None*, *relabel_nodes=True*)

Convert an FNSS topology to Mininet Topo object that can be used to deploy a Mininet network.

If the links of the topology are labeled with delays, capacities or buffer sizes, the returned Mininet topology will also include those parameters.

However, it should be noticed that buffer sizes are included in the converted topology only if they are expressed in packets. If buffer sizes are expressed in the form of bytes they will be discarded. This is because Mininet only supports buffer sizes expressed in packets.

Parameters

topology [Topology, DirectedTopology or DatacenterTopology] An FNSS Topology object

switches [list, optional] List of topology nodes acting as switches

hosts [list, optional] List of topology nodes acting as hosts

relabel_nodes [bool, optional] If *True*, rename node labels according to Mininet conventions. In Mininet all node labels are strings whose values are “h1”, “h2”, … if the node is a host or “s1”, “s2”, … if the node is a switch.

Returns

topology [Mininet Topo] A Mininet topology object

Notes

It is not necessary to provide a list of switch and host nodes if the topology object provided are already annotated with a type attribute that can have values *host* or *switch*. This is the case of datacenter topologies generated with FNSS which already include information about which nodes are hosts and which are switches.

If switches and hosts are passed as arguments, then the hosts and switches sets must be disjoint and their union must coincide to the set of all topology nodes. In other words, there cannot be nodes labeled as both *host* and *switch* and there cannot be nodes that are neither a *host* nor a *switch*.

It is important to point out that if the topology contains loops, it will not work with the *ovs-controller* and *controller* provided by Mininet. It will be necessary to use custom controllers. Further info here.

`ns2` module

Adapter for ns-2.

This module contains the code for converting an FNSS topology object into a Tcl script to deploy such topology into ns-2.

`to_ns2`(topology, path[, stacks])	Convert topology object into an ns-2 Tcl script that deploys that topology into ns-2.
`validate_ns2_stacks`(topology)	Validate whether the stacks and applications of a topology are valid for ns-2 deployment

fnss.adapters.ns2.to_ns2

`to_ns2` (*topology*, *path*, *stacks=True*)

Convert topology object into an ns-2 Tcl script that deploys that topology into ns-2.

Parameters

topology [Topology] The topology object to convert

path [str] The path to the output Tcl file

stacks [bool, optional] If True, read the stacks on nodes and write them into the output file. For this to work, stacks must be formatted in a way understandable by ns-2. For example, stack and applications must have a 'class' attribute whose value is the name of the ns-2 class implementing it.

Notes

In order for the function to parse stacks correctly, the input topology must satisfy the following requirements:

- each stack and each application must have a *class* attribute whose value is the ns-2 class implementing such stack or application, such as *Agent/TCP* or *Application/FTP*.
- All names and values of stack and application properties must be valid properties recognized by the ns-2 application or protocol stack.

fnss.adapters.ns2.validate_ns2_stacks

validate_ns2_stacks(*topology*)

Validate whether the stacks and applications of a topology are valid for ns-2 deployment

Parameters

topology [Topology] The topology object to validate

Returns

valid [bool] *True* if stacks are valid ns-2 stacks, *False* otherwise

omnetpp module

Omnet++ adapter

This module contains the code for converting an FNSS topology object into a NED script to deploy such topology into Omnet++.

to_omnetpp(topology[, path])	Convert an FNSS topology into an Omnet++ NED script.

fnss.adapters.omnetpp.to_omnetpp

to_omnetpp(*topology*, *path=None*)

Convert an FNSS topology into an Omnet++ NED script.

Parameters

topology [Topology] The topology object to convert

path [str, optional] The path to the output NED file. If not specified, prints to standard output

1.3.3 Scripts

1.3.3.1 mn-fnss

Usage:

```
mn-fnss [mn-options] [--no-relabel] <topology-file>
mn-fnss (--help | -h)
mn-fnss (--version | -v)
```

Options:

```
mn-options         Mininet mn options.
--no-relabel       Do not relabel topology nodes to Mininet conventions.
-h --help          Show help.
-v --version       Show version.
```

Launch Mininet console with an FNSS topology.

This script parses an FNSS topology XML file and launches the Mininet console passing this topology.

This script accepts all the options of Mininet *mn* script, except for the *custom* and *topo* options which are overwritten by this script.

In addition, if the user specifies the mn *link* option, then all potential link attributes of the topology (e.g. capacity, delay and max queue size) are discarded and values provided with the link attributes are used instead.

Unless the option *–no-relabel* is provided, this script relabels all nodes of the FNSS topology to match Mininet's conventions, i.e. each host label starts with *h* (e.g. h1, h2, h3…) and each switch label starts with *s* (e.g. s1, s2, s3…).

Unless used to print this help message or version information, this script must be run as superuser.

Example usage:

```
$ python
>>> import fnss
>>> topo = fnss.two_tier_topology(1, 2, 2)
>>> fnss.write_topology(topo, 'fnss-topo.xml')
$ sudo mn-fnss fnss-topo.xml
```

1.3.3.2 fnss-troubleshoot

Usage:

```
fnss-troubleshoot [--help | -h]
```

This script prints debugging information about FNSS dependencies currently installed.

The main purpose of this script is to help users to communicate effectively with developers when reporting an issue.

CHAPTER 2

Indices and tables

- genindex
- modindex

Bibliography

[3] Nucci et al., The problem of synthetically generating IP traffic matrices: initial recommendations, ACM SIGCOMM Computer Communication Review, 35(3), 2005

[1] Nucci et al., The problem of synthetically generating IP traffic matrices: initial recommendations, ACM SIGCOMM Computer Communication Review, 35(3), 2005

[2] Nucci et al., The problem of synthetically generating IP traffic matrices: initial recommendations, ACM SIGCOMM Computer Communication Review, 35(3), 2005

[1] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: a high performance, host-centric network architecture for modular data centers. Proceedings of the ACM SIGCOMM 2009 conference on Data communication (SIGCOMM '09). ACM, New York, NY, USA. http://doi.acm.org/10.1145/1592568.1592577

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. Proceedings of the ACM SIGCOMM 2008 conference on Data communication (SIGCOMM '08). ACM, New York, NY, USA http://doi.acm.org/10.1145/1402958.1402967

[1] A. L. Barabasi and R. Albert "Emergence of scaling in random networks", Science 286, pp 509-512, 1999.

[1] 16. Erdos and A. Renyi, On Random Graphs, Publ. Math. 6, 290 (1959).

[2] 5. (n) Gilbert, Random Graphs, Ann. Math. Stat., 30, 1141 (1959).

[3] Vladimir Batagelj and Ulrik Brandes, "Efficient generation of large random networks", Phys. Rev. E, 71, 036113, 2005.

[1] A. L. Barabasi and R. Albert "Topology of evolving networks: local events and universality", Physical Review Letters 85(24), 2000.

[1] T. Bu and D. Towsey "On distinguishing between Internet power law topology generators", Proceeding od the 21st IEEE INFOCOM conference. IEEE, volume 2, pages 638-647, 2002.

[1] B. M. Waxman, Routing of multipoint connections. IEEE J. Select. Areas Commun. 6(9),(1988) 1617-1622.

[1] B. M. Waxman, Routing of multipoint connections. IEEE J. Select. Areas Commun. 6(9),(1988) 1617-1622.

[1] I. Stoica, R. Morris, D. Karger, M. Frans Kaashoek, H. Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. Proceedings of the ACM SIGCOMM 2001 conference on Data communication (SIGCOMM '09). ACM, New York, NY, USA.

Python Module Index

f

Index

V

W