pyNTM

Release 3.4.1

Oct 12, 2021

Contents

This is a network traffic modeler written in python 3. This library allows users to define a layer 3 network topology, define a traffic matrix, and then run a simulation to determine how the traffic will traverse the topology. If you've used Cariden MATE or WANDL, this code solves for some of the same basic use cases those do.

CHAPTER 1

Typical Use Cases

- How will a failure on your wide area network (WAN) affect link utilizations?
- What about a layer 3 node failure?
- Will all of your RSVP auto-bandwidth LSPs be able to resignal after a link/node failure?
- Can you WAN handle a 10% increase in traffic during steady state? What about failover state?

These questions are non-trivial to answer for a medium to large size WAN that is meshy/interconnected, and these are the exact scenarios that a WAN simulation engine is designed to answer.

Changes to the topology can be done to simulate new routers, circuits, circuit capacity, network failures, etc. Changes to the traffic matrix can be done to simulate increases/decreases in existing traffic or additional traffic matrix entries.

This is the network traffic modeler written in python 3. The main use cases involve understanding how your layer 3 traffic will transit a given topology. You can modify the topology (add/remove layer 3 Nodes, Circuits, Shared Risk Link Groups), fail elements in the topology, or add new traffic Demands to the topology. pyNTM is a simulation engine that will converge the modeled topology to give you insight as to how traffic will transit a given topology, either steady state or with failures.

This library allows users to define a layer 3 network topology, define a traffic matrix, and then run a simulation to determine how the traffic will traverse the topology, traverse a modified topology, and fail over. If you've used Cariden MATE or WANDL, this code solves for some of the same basic use cases those do. This package is in no way related to those, or any, commercial products. IGP and RSVP routing is supported.

pyNTM can be used as an open source solution to answer WAN planning questions; you can also run pyNTM alongside a commercial solution as a validation/check on the commercial solution.

CHAPTER 2

Example Scripts and Training

See the [example directory](https://github.com/tim-fiola/network_traffic_modeler_py3/blob/master/examples).

Examine and run the client code examples, these docs, and check out the pyNTM training repository to get an understanding of how to use this code and the use cases.

CHAPTER 3

Full API set use cases include

- simulating how traffic will transit your layer 3 network, given a traffic matrix
- simulating how your traffic will failover if any link(s) or node(s) fail
- simulating how a given network augment will affect link utilization and traffic flow, possible augments include
  - adding a new link
  - adding a new node with links
  - changing a link's capacity
  - adding a new traffic matrix entry (demand) to the traffic matrix
  - increasing/decreasing the magnitude of an existing demand in the traffic matrix failover

Note: interface circuit_ids are only used to match interfaces into circuits and do not have any practical bearing on the simulation results

CHAPTER 4

Contents

4.1 Install

To use the library as a module, it is recommended to install it via pip as shown below.

Install via pip:

```
$ pip3 install pyNTM
```

To upgrade:

```
$ pip3 install --upgrade pyNTM
```

4.2 Examples

4.2.1 Demo/Example scripts

These scripts are in the `examples` directory and are meant to showcase different capabilities and how to use them.

The `lsp_practice_code.py` script demos how auto-bandwidth RSVP LSPs react to

- link failures
- adding traffic
- adding additional LSPs

The `network_modeling_client_code_examples_read_from_dict.py` file demo2 the following:

- loading a network topology from a list of info (instead of a model file)
- addition of new circuit and node to the network
- viewing interface traffic

- getting the shortest path
- failing an interface
- demand path changes before/after a link failure
- adding traffic

4.2.2 Demo Script With Visualization

There is also a visualization script, which showcases a current beta feature. See the visualization docs page for more details.

4.3 The Model Object

4.3.1 What is the model object?

The model object has two main functions:

- It holds all the objects that represent the network (interfaces, nodes, demands, RSVPs, etc)
- It controls the mechanics of how each object interacts with others

A network model requires these primitive objects to create a simulation and higher-level objects:

- Interfaces
- Nodes
- Demands
- RSVP LSPs (only required if network has RSVP LSPs)

From these primitives, the model can construct higher-level objects such as:

- Circuits
- Shared Risk Link Groups (SRLGs)
- Paths for demands and LSPs
- etc

The model produces a simulation of the network behavior by applying the traffic demand objects to the network topology, routing the traffic demands across the topology as a real network would. The model produces a simulation when its `update_simulation()` method is called.

4.3.2 Model Type Subclasses

There are two network model subclasses: `FlexModel` and `PerformanceModel`.

In general, the `FlexModel` can accommodate more topology variations, but at the price of a slightly longer convergence time while the `PerformanceModel` can only handle simpler network architectures, but with the benefit of better convergence time.

All model classes support:

- IGP routing
- RSVP LSPs carrying traffic demands that have matching source and destination as the RSVP LSPs

- RSVP auto-bandwidth or fixed bandwidth
- RSVP LSP manual metrics

The `PerformanceModel` class allows for:

- Single circuits between 2 Nodes
- Error messages if it detects use of IGP shortcuts or multiple Circuits between 2 Nodes

The `FlexModel` class allows for:

- Multiple Circuits between 2 Nodes
- RSVP LSP IGP shortcuts, whereby LSPs can carry traffic demands downstream, even if the demand does not have matching source and destination as the LSP

4.3.3 How do I know the simulations work correctly?

There are many safeguards in place to ensure the simulation's mechanics are correct:

- Multiple functional tests in the CI/CD pipeline check for the expected mechanics and results for each routing method (ECMP, single path, RSVP, RSVP resignaling, etc) and other features in various topology scenarios:
  - If any of these tests fail, the build will fail and the bad code will not make it into production
    - This helps ensure that functionality works as expected and that new features and fixes don't break prior functionality
  - There are over 300 unit and functional tests in the pyNTM CI/CD pipeline
  - There are dozens of topology-specific functional tests in the pyNTM CI/CD pipeline that ensure the simulation works properly for different topologies, and more are added for each new feature
- The model object has internal checks that happen automatically during the `update_simulation()` execution:
  - Flagging interfaces that are not part of a circuit
  - Flagging if an interface's RSVP reserved bandwidth is greater than the interface's capacity
  - Verifying that each interface's RSVP reserved bandwidth matches the sum of the reserved bandwidth for the LSPs egressing the interface
  - Checks that the interface names on each node are unique
  - Validates that the capacities of each of the component interfaces in a circuit match
  - Validates that each node in an SRLG has the SRLG in the node's `srlgs` set
  - No duplicate node names are present in the topology

The PerformanceModel subclass also verifies that

- IGP shortcuts are not enabled for nodes
- There is no more than a single circuit (edge) between any two nodes

Note that there are more checks involving RSVP than IGP/ECMP routing because there are more mechanics involved when RSVP is running, whereas the straight IGP/ECMP routing is much simpler.

If any of these checks fails, `update_simulation()` will throw an error with debug info.

4.4 The Network Model File

The network model file contains basic information about the network topology:

- Interfaces
- Nodes (layer 3 node/router)
- Demands (traffic)
- RSVP LSPs

Since there is a lot of information needed to create a model, using the model file to load network data is recommended.

The network model file is a **tab-separated** values file.

The sections below describe the model file's table headers for the FlexModel and PerformanceModel subclasses.

4.4.1 Model File Interface Headers

Interfaces represent the logical interfaces on a layer 3 Node (Router). In the context of a simulation, demands (traffic) egress interfaces. A circuit is created when two *matching* interfaces (one in each direction) are paired to form a bi-directional connection between layer 3 nodes.

The process of *matching* circuits depends on the model object used.

Since there can only be a single connection (circuit) between any two nodes, the `PerformanceModel` automatically matches the interfaces into circuits.

The `FlexModel` requires a `circuit_id` to appropriately match two interfaces into a circuit. The `circuit_id` must be included for each interface in the model file's `INTERFACES_TABLE`. There must be exactly two instances of a given `circuit_id` in the `INTERFACES_TABLE`: any more or any less will result in extra/unmatched interfaces and will cause an error when the file is loaded.

PerformanceModel Interfaces

INTERFACES_TABLE

- `node_object_name` - name of node where interface resides
- `remote_node_object_name` - name of remote node
- `name` - interface name
- `cost` - IGP cost/metric for interface
- `capacity` - capacity
- `rsvp_enabled` (optional) - is interface allowed to carry RSVP LSPs? True|False; default is True
- `percent_reservable_bandwidth` (optional) - percent of capacity allowed to be reserved by RSVP LSPs; this value should be given as a percentage value - ie 80% would be given as 80, NOT .80. Default is 100

FlexModel Interfaces

INTERFACES_TABLE

- `node_object_name` - name of node where interface resides
- `remote_node_object_name` - name of remote node

- `name` - interface name
- `cost` - IGP cost/metric for interface
- `capacity` - capacity
- `circuit_id` - id of the circuit; used to match two Interfaces into Circuits
  - each `circuit_id` value can only appear twice in the model
  - `circuit_id` can be string or integer
- `rsvp_enabled` (optional) - is interface allowed to carry RSVP LSPs? True|False; default is True
- `percent_reservable_bandwidth` (optional) - percent of capacity allowed to be reserved by RSVP LSPs; this value should be given as a percentage value - ie 80% would be given as 80, NOT .80. Default is 100
- `manual_metric` (optional) - manually assigned metric for LSP, if not using default metric from topology shortest path

Important: Column order matters. If you wish to use an optional column to the right of an optional column you don't want to specify a value for, you must still include the optional headers to the left of the column you wish to specify a value for.

For example, if you wish to specify a `percent_reservable_bandwidth` for an interface but not explicitly specify `rsvp_enabled`, you must also include the `rsvp_enabled` columns and then leave those row values blank in each unused column.

This example specifies `percent_reservable_bandwidth` of 30 for interface `A-to-B_1`:

```
INTERFACES_TABLE
node_object_name    remote_node_object_name name    cost    capacity    circuit_id ␣
↪rsvp_enabled    percent_reservable_bandwidth
A   B       A-to-B_1    20  120 1       30
B   A       B-to-A_1    20  120 1   True  50
```

4.4.2 Model File Node Headers

Nodes represent layer 3 devices in the topology. Many nodes can be inferred by the presence of an interface on the `node_object` column in the `INTERFACES_TABLE` in the model file. Any node inferred by the `node_object` column in the `INTERFACES` table does not have to be explicitly declared in the `NODES` table. However, the `NODES` table does have a couple of use cases:

- It can be used to add attributes to inferred nodes: `lat` (latitude, or y-coordinate), `lon` (longitude, or x-coordinate), and `igp_shortcuts_enabled` (whether IGP shortcuts are enabled for the node)
- It can be used to declare a node that does not have any interfaces yet (aka an *orphan* node)

Note: `lat` and `lon` can be used instead for (y, x) grid coordinates; there are no restrictions on the integer values those attributes can have.

PerformanceModel Nodes

NODES_TABLE

- `name` - name of node
- `lon` - longitude (or y-coordinate) (optional)
- `lat` - latitude (or x-coordinate) (optional)

FlexModel Nodes

NODES_TABLE

- `name` - name of node
- `lon` - longitude (or y-coordinate) (optional)
- `lat` - latitude (or x-coordinate) (optional)
- `igp_shortcuts_enabled` (default=‘‘False‘‘) - Indicates if IGP shortcuts enabled for the Node * If `True`, network internal traffic transiting the layer 3 node can now use LSPs en route to the destination, if they are available

Important: Column order matters. If you wish to use an optional column to the right of an optional column you don’t want to specify a value for, you must still include the optional headers to the left of the column you wish to specify a value for.

If you wish to include `igp_shortcuts_enabled` values for a given node, you must include the `name`, `lon` and `lat` column headers and then leave the unused row values for those columns blank.

For example, to enable `igp_shortcuts_enabled` for the `SLC` node, but not specify `lon` or `lat`:

```
NODES_TABLE
name lon lat igp_shortcuts_enabled
SLC           True
```

4.4.3 Model File Demand Headers

Demands represent traffic on the network. Each demand represents an amount of traffic ingressing the network at a specific layer 3 (source) node and egressing the network at a specific layer 3 (destination) node.

PerformanceModel and FlexModel Demands

For both model classes, the `DEMANDS_TABLE` table has four headers, all of which are required:

- `source` - the source node for the traffic; the node in the model where the traffic originates
- `dest` - the destination node for the traffic; the node in the model where the traffic terminates
- `traffic` - the amount of traffic in the demand
- `name` - the name of the demand; there can be multiple demands with matching source and dest nodes - the name is the differentiator
  - there cannot be multiple demands with matching `source`, `dest`, and `name` values

4.4.4 RSVP LSPs

PerformanceModel and FlexModel RSVP LSPs

The RSVP_LSP_TABLE has the following columns:

- source - the source node for the LSP; the node in the model where the LSP originates
- dest - the destination node for the LSP; the node in the model where the LSP terminates
- name - the name of the LSP; there can be multiple LSPs with matching source and dest nodes - the name is the differentiator
  - There cannot be multiple LSPs with matching source, dest, and name values
- configured_setup_bw (optional) - if LSP has a fixed, static configured setup bandwidth, place that static value here, if LSP is auto-bandwidth, then leave this blank for the LSP
- manual_metric (optional) - manually assigned metric for LSP, if not using default metric from topology shortest path

Important: Column order matters. If you wish to use an optional column to the right of an optional column you don't want to specify a value for, you must still include the optional headers to the left of the column you wish to specify a value for.

If you wish to specify a manual_metric for an LSP but not explicitly specify configured_setup_bw, you must also include the configured_setup_bw column and then leave those row values blank in each unused column.

For example, to specify a manual_metric for the LSP with name lsp_a_b_2 but not specify configured_setup_bw:

```
RSVP_LSP_TABLE
source      dest    name    configured_setup_bw manual_metric
A   B       lsp_a_b_1   10  19
A   B       lsp_a_b_2       6
```

4.5 RSVP LSP

A class to represent an RSVP label-switched-path in the network model

source_node_object: Node where LSP ingresses the network (LSP starts here)

dest_node_object: Node where LSP egresses the network (LSP ends here)

lsp_name: name of LSP

path:

```
will either be 'Unrouted' or be a dict containing the following -
- interfaces: list of interfaces that LSP egresses in the order it
    egresses them
- path_cost: sum of costs of the interfaces
- baseline_path_reservable_bw: the amount of reservable bandwidth
    available on the LSP's path when the LSP was signaled, not inclusive
    of the bandwidth already reserved by this LSP on that path (if any)
```

reserved_bandwidth: amount of bandwidth reserved by this LSP

setup_bandwidth: amount of bandwidth this LSP attempts to signal for when it sets up

manual_metric: manual metric for LSP. If set, this value will override the default (shortest path) metric for effective_metric. This value must be a positive integer. To restore the LSP's default metric (that of the shortest IGP path) in a live simulation, set this value to -1.

For more information on the capabilities of RSVP LSPs, see the RSVP LSP docstrings.

4.6 Demands

A representation of traffic load on the modeled network.

A demand object must have a source node, a destination node, and a name.

4.6.1 Demand Units

A demand's `traffic` property shows how many traffic *units* the demand carries. The `traffic` property is unit-less, meaning it is not described in Mbps, Gbps, etc. It is written in this generic way so that the user can determine the traffic units they want to deal with in their modeling exercises.

For example, if a demand has `traffic` = 100, it can be 100Mpbs, 100Gbps, etc. This example, will use Gbps.

If the entire demand transits a single interface (# ECMP = 1) with a `capacity` of 200 Gbps (and no other demands transit the interface), the interface's computed `utilization` will be 50%.

A demand's path is

4.6.2 The `path_detail` Property

The demand object's `path_detail` property can be very useful to determine how much of the demand's traffic egresses each object (interface, LSP) in the path:

```
Returns a detailed breakdown of the Demand path.
Each path will have the following information:

items: The combination of Interfaces and/or LSPs that the Demand transits
from source to destination

splits: each item on the path (Interface and/or LSP) and the number of cumulative
ECMP path splits that the Demand has transited as it egresses the source node for
that element.

path_traffic: the amount of traffic on that specific path for the demand.  Path␣
↪traffic will be the
result of dividing the Demand's traffic by the **max** amount of path splits for an
element in the path
```

For example, sample demand `Demand(source = A, dest = E, traffic = 24, name = 'dmd_a_e_1')` has 24 units of traffic.

Here is the `path_0` entry for the sample demand:

```
'path_0': {
    'items': [Interface(name = 'A-to-B', cost = 4, capacity = 100, node_object = Node(
↪'A'),
                remote_node_object = Node('B'), circuit_id = '1'),
            Interface(name = 'B-to-E_3', cost = 3, capacity = 200, node_object = Node(
↪'B'),
                remote_node_object = Node('E'), circuit_id = '27')
    ],
    'path_traffic': 4.0,
    'splits': {Interface(name = 'A-to-B', cost = 4, capacity = 100, node_object =␣
↪Node('A'),
                remote_node_object = Node('B'), circuit_id = '1'): 2,
              Interface(name = 'B-to-E_3', cost = 3, capacity = 200, node_object =␣
↪Node('B'),
                remote_node_object = Node('E'), circuit_id = '27'): 6}
}
```

The `path_0` component of the `path_detail` property in this example shows the following:

- `Interface(name = 'A-to-B', cost = 4, capacity = 100, node_object = Node('A'), remote_node_object = Node('B'), circuit_id = '1')` has **2** splits
- `Interface(name = 'B-to-E_3', cost = 3, capacity = 200, node_object = Node('B'), remote_node_object = Node('E'), circuit_id = '27')` has **6** splits

To get the amount of traffic load from the specific demand that transits each interface, divide the amount of traffic that the demand has by the number of splits for the object:

- `Interface(name = 'A-to-B', cost = 4, capacity = 100, node_object = Node('A'), remote_node_object = Node('B'), circuit_id = '1')` carries **24 / 2 = 12** units of traffic from `Demand(source = A, dest = E, traffic = 24, name = 'dmd_a_e_1')`
- `Interface(name = 'B-to-E_3', cost = 3, capacity = 200, node_object = Node('B'), remote_node_object = Node('E'), circuit_id = '27')` carries **24 / 6 = 4** units of traffic from `Demand(source = A, dest = E, traffic = 24, name = 'dmd_a_e_1')`

Since the minimum amount of traffic found on any object in `path_0` is 4 units of traffic, `path_traffic` for `path_0` = 4.

For more information on demands,see the demand docstrings.

Please see the pyNTM Training Modules repository module 2 for info and walk-through exercises for demands, including:

- Finding traffic demands egressing a given interface
- Finding all ECMP paths for a specific demand

4.7 Interfaces

Interfaces represent uni-directional interfaces on a layer 3 node.

A **circuit** is created when two *matching* interfaces (one in each direction) are paired to form a bi-directional connection between layer 3 nodes.

Interfaces are matched into a circuit if:

- They are on different nodes
- They have the same `capacity`
- They have a matching `circuit_id`

Important: In the `PerformanceModel` subclass, the `circuit_id` is hidden because this subclass only allows a single circuit (edge) between any two nodes, so matching interfaces into circuits is deterministic.

Interface `utilization` represents how much traffic is egressing the interface divided by the interface's `capacity` (`utilization` = `traffic` / `capacity`).

4.7.1 Interface Capacity Units

An interface's `capacity` and `traffic` are described in *units* of traffic, meaning they are not described in Mbps, Gbps, etc. They are written in this generic way so that the user can determine the traffic units they want to deal with in their modeling exercises.

For example, if an interface has `capacity` = 200, it can be 200 Mpbs, 200 Gbps, etc. This example, will use Gbps.

If a single, entire demand (# ECMP = 1) with `traffic` = 100 (Gbps) transits the interface, the interface's computed `utilization` will be 50% and its `traffic` will be 100 (Gbps).

For more information on interfaces,see the interface docstrings.

4.8 Shared Risk Link Groups (SRLGs)

An SRLG represents a collection of model objects with shared risk factors.

An SRLG can include:

- Nodes
- Circuits

When the SRLG is failed (`failed = True`), the members will go to a failed state (`failed = True`) as well. When the SRLG is not failed (`failed = False`), the members will also return to `failed = False`.

Nodes are added to an SRLG via the `add_to_srlg` node method.

A circuit is added to an SRLG when a component interface is added to the SRLG via the `add_to_srlg` interface method. The other interface in the interface's circuit is also automatically added to the SRLG.

4.9 Common Workflows

There is an existing pyNTM training repository on GitHub that extensively covers common workflows with pyNTM.

pyNTM Training Modules repository module 1 covers what network modeling is, the problem it solves for, and the common use cases

Please see the pyNTM Training Modules repository module 2 for info and walk-through exercises for

- Directions on how to get started using pyNTM
- Setting up a practice/demo environment
- Finding Shortest Path(s)

- Failing/Unfailing Interfaces
- Finding traffic demands egressing a given interface
- Finding all ECMP paths for a specific demand
- Simple visualization exercise

Please see the pyNTM Training Modules repository module 3 for info and walk-through exercises for

- Adding a new Node
- Adding a new link
- Adding traffic to the traffic matrix
- Changing Interface/Circuit capacity
- Changing an Interface metric
- Working with RSVP LSPs

Please see the pyNTM Training Modules repository module 4 for info and walk-through exercises for

- RSVP LSP model data files
- RSVP types and behaviors
- Auto bandwidth
- Fixed bandwidth
- LSPs and Demands
- Getting an LSP path
- Seeing demands on an LSP
- Demand path when demand is on LSP
- Shared Risk Link Groups (SRLGs)
- Adding an SRLG
- Failing an SRLG

Please see the pyNTM Training Modules repository module 5 for info and walk-through exercises for

- How to create a visualization using the WeatherMap
- WeatherMap visual components overview

4.9.1 Checking Network Health

There are a some results to watch for in your simulations that will indicate a network augment or re-architecture of your existing or planned network may be helpful.

IGP routing is deterministic and much simpler to interpret; one obvious warning sign is over-utilized links.

It gets a bit more difficult with RSVP, especially with auto-bandwidth enabled, to determine if the network is under stress. RSVP auto-bandwidth behavior can be non-deterministic, meaning that there may be multiple different end-states the network will converge to, depending on the order in which the LSPs signal and how long each layer 3 node takes to compute the paths for its LSPs and a host of other factors.

With this being the case, there are a few behavior in the model to watch for when running RSVP that may indicate a network augment or re-architecture may be helpful:

- Large quantities of LSPs not on the shortest path

- LSPs reserving less bandwidth than they are carrying
- Some LSPs not being able to signal due to lack of available setup bandwidth in the path

4.10 Visualization

PyNTM has a visualization feature that produces an interactive visualization of the network.

More info on the visualization and how to use it can be found in the pyNTM training repository visualization training module.

4.10.1 Visualization Demo Script

The `examples/vis_training.py` example script showcases the visualization module, which is current beta feature.

To run this script, install the packages in `requirements_visualization.txt` file:

```
(venv) % pip3 install -r requirements_visualization.txt
```

Once you've installed the requirements, run the script.

You will see some simulation activities and info and then a visualization:

```
(venv) % python3 -i vis_training.py
Routing the LSPs . . .
Routing 1 LSPs in parallel LSP group C-E; 1/3
Routing 1 LSPs in parallel LSP group D-F; 2/3
Routing 2 LSPs in parallel LSP group B-D; 3/3
LSPs routed (if present) in 0:00:00.001134; routing demands now . . .
Demands routed in 0:00:00.003497; validating model . . .
shortest path from A to F is:
{'cost': 50,
 'path': [[Interface(name = 'A-G', cost = 25, capacity = 100, node_object = Node('A'),
↪ remote_node_object = Node('G'), circuit_id = '6'),
           Interface(name = 'G-F', cost = 25, capacity = 100, node_object = Node('G'),
↪ remote_node_object = Node('F'), circuit_id = '7')],
          [Interface(name = 'A-G', cost = 25, capacity = 100, node_object = Node('A'),
↪ remote_node_object = Node('G'), circuit_id = '6'),
           Interface(name = 'G-F_2', cost = 25, capacity = 100, node_object = Node('G
↪'), remote_node_object = Node('F'), circuit_id = '8')],
          [Interface(name = 'A-B', cost = 10, capacity = 100, node_object = Node('A'),
↪ remote_node_object = Node('B'), circuit_id = '1'),
           Interface(name = 'B-C', cost = 10, capacity = 100, node_object = Node('B'),
↪ remote_node_object = Node('C'), circuit_id = '2'),
           Interface(name = 'C-D', cost = 10, capacity = 100, node_object = Node('C'),
↪ remote_node_object = Node('D'), circuit_id = '3'),
           Interface(name = 'D-E', cost = 10, capacity = 100, node_object = Node('D'),
↪ remote_node_object = Node('E'), circuit_id = '4'),
           Interface(name = 'E-F', cost = 10, capacity = 100, node_object = Node('E'),
↪ remote_node_object = Node('F'), circuit_id = '5')],
          [Interface(name = 'A-B', cost = 10, capacity = 100, node_object = Node('A'),
↪ remote_node_object = Node('B'), circuit_id = '1'),
           Interface(name = 'B-C_2', cost = 10, capacity = 100, node_object = Node('B
↪'), remote_node_object = Node('C'), circuit_id = '9'),
           Interface(name = 'C-D', cost = 10, capacity = 100, node_object = Node('C'),
↪ remote_node_object = Node('D'), circuit_id = '3'),
```

(continues on next page)

(continued from previous page)

```
          Interface(name = 'D-E', cost = 10, capacity = 100, node_object = Node('D'),
↪ remote_node_object = Node('E'), circuit_id = '4'),
          Interface(name = 'E-F', cost = 10, capacity = 100, node_object = Node('E'),
↪ remote_node_object = Node('F'), circuit_id = '5')]]}

dmd_a_f_1 path is:
[[Interface(name = 'A-G', cost = 25, capacity = 100, node_object = Node('A'), remote_
↪node_object = Node('G'), circuit_id = '6'),
  Interface(name = 'G-F', cost = 25, capacity = 100, node_object = Node('G'), remote_
↪node_object = Node('F'), circuit_id = '7')],
 [Interface(name = 'A-G', cost = 25, capacity = 100, node_object = Node('A'), remote_
↪node_object = Node('G'), circuit_id = '6'),
  Interface(name = 'G-F_2', cost = 25, capacity = 100, node_object = Node('G'),␣
↪remote_node_object = Node('F'), circuit_id = '8')],
 [Interface(name = 'A-B', cost = 10, capacity = 100, node_object = Node('A'), remote_
↪node_object = Node('B'), circuit_id = '1'),
  RSVP_LSP(source = B, dest = D, lsp_name = 'lsp_b_d_1'),
  RSVP_LSP(source = D, dest = F, lsp_name = 'lsp_d_f_1')],
 [Interface(name = 'A-B', cost = 10, capacity = 100, node_object = Node('A'), remote_
↪node_object = Node('B'), circuit_id = '1'),
  RSVP_LSP(source = B, dest = D, lsp_name = 'lsp_b_d_2'),
  RSVP_LSP(source = D, dest = F, lsp_name = 'lsp_d_f_1')]]

*** NOTE: The make_visualization_beta function is a beta feature.  It may not have␣
↪been as
extensively tested as the pyNTM code in general.  The API calls for this may also
change more rapidly than the general pyNTM code base.



Visualization is available at http://127.0.0.1:8050/


Dash is running on http://127.0.0.1:8050/

 * Serving Flask app 'pyNTM.weathermap' (lazy loading)
 * Environment: production
   WARNING: This is a development server. Do not use it in a production deployment.
   Use a production WSGI server instead.
 * Debug mode: off
 * Running on http://127.0.0.1:8050/ (Press CTRL+C to quit)
127.0.0.1 - - [26/Sep/2021 18:18:19] "GET / HTTP/1.1" 200 -
127.0.0.1 - - [26/Sep/2021 18:18:20] "GET /_dash-layout HTTP/1.1" 200 -
127.0.0.1 - - [26/Sep/2021 18:18:20] "GET /_dash-dependencies HTTP/1.1" 200 -
127.0.0.1 - - [26/Sep/2021 18:18:20] "GET /_favicon.ico?v=2.0.0 HTTP/1.1" 200 -
127.0.0.1 - - [26/Sep/2021 18:18:20] "GET /_dash-component-suites/dash/dcc/async-
↪dropdown.js HTTP/1.1" 200 -
127.0.0.1 - - [26/Sep/2021 18:18:20] "POST /_dash-update-component HTTP/1.1" 200 -
127.0.0.1 - - [26/Sep/2021 18:18:20] "POST /_dash-update-component HTTP/1.1" 200 -
```

If you open your browser to http://127.0.0.1:8050/, you will see an interactive visualization.

4.11 Development

If you wish to contribute PRs to pyNTM, the sections below describe how to set up your development environment.

Currently, pyNTM adheres to the *black* code formatting and *flake8*. More info on *black* is below.

The *black* code formatting and *flake8* checks occur in the CI/CD pipeline.

The sections below cover how to set up your local dev environment to run these checks prior to submitting a pull request.

4.11.1 Set Up Your Virtual Environment

Set up your virtual environment.

Virtualenv is a tool for creating isolated 'virtual' python environments. For directions on how to run this modeler in a virtual environment and auto-download all dependencies, follow the steps below (authored by nelsg).

Create your virtualenv

Create an isolated virtual environment under the directory "network_traffic_modeler_py3" with python3:

```
$ virtualenv -p python3 venv
```

Activate "venv" that sets up the required env variables:

```
$ source venv/bin/activate
```

Clone the repo per https://docs.github.com/en/repositories/creating-and-managing-repositories/cloning-a-repository#cloning-a-repository:

```
(venv) % git clone https://github.com/tim-fiola/network_traffic_modeler_py3.git
Cloning into 'network_traffic_modeler_py3'...
remote: Enumerating objects: 1622, done.
remote: Counting objects: 100% (378/378), done.
remote: Compressing objects: 100% (256/256), done.
remote: Total 1622 (delta 247), reused 214 (delta 120), pack-reused 1244
Receiving objects: 100% (1622/1622), 770.31 KiB | 1.89 MiB/s, done.
Resolving deltas: 100% (1107/1107), done.
(venv)  %
```

Install pyNTM's required packages with "pip3":: (venv) % cd network_traffic_modeler_py3 (venv) % pip3 install -r requirements.txt

4.11.2 Installing Development Requirements

To install the Python modules required for development, run:

```
% pip3 install -r requirements_dev.txt
```

The Black Code Formatter

pyNTM uses the black code formatter. `black` is an opinionated formatter that will make changes to your files to bring them in line with the `black` standards.

Setting Up Pre-Commit Hooks

The `pre-commit` package is installed when you install the `requirements_dev.txt`.

To run `black` automatically after each commit, you will need to install the pre-commit hooks.

Run `pre-commit install`:

```
% pre-commit install
```

You should see a response that the hook has been installed:

```
pre-commit installed at .git/hooks/pre-commit
```

This will set up a check that runs `black` prior to allowing a commit, allowing you to focus on making your code, instead of worrying about your formatting.

4.11.3 Local Unit Testing

To run the unit tests locally:

1. Go to the repository's `network_traffic_modeler_py3` directory:

   ```
   (venv) % pwd
   /path/to/network_traffic_modeler_py3
   (venv) %
   ```

2. Run `pytest`:

   ```
   ``% pytest``
   ```

If the tests fail to run due to `ImportError`, depending on your OS, you may need to run one of the following `pytest` variations:

```
``python -m pytest``
```

or:

```
``python3 -m pytest``
```

Tip: When submitting a pull request, your build will be tested against black and the unit tests, so it's advantageous to test them locally prior so they don't fail in the CI pipeline.

4.11.4 Remove your virtualenv

If you wish to remove your virtualenv when you are complete, follow the steps below.

Deactivate "venv" that unsets the virtual env variables:

```
$ deactivate
```

Remove directory "venv":

```
$ rm -rf venv
```

4.11.5 pypy3

pyNTM is compatible with the pypy3 interpreter. The pypy3 interpreter provides a 70-80% performance improvement over the python3 interpreter.

Tip: By *performance*, we mean the time it takes to converge the model to produce a simulation (running `update_simulation`).

It is recommended, however, to *develop* in **python3**. Developing in **pypy3** is **NOT** recommended, because some of the developer tools are not compatible in a pypy3 environment.

4.12 API

4.12.1 _MasterModel

class pyNTM.master_model.**_MasterModel**(*interface_objects={}*, *node_objects={}*, *demand_objects={}*, *rsvp_lsp_objects={}*)

Bases: `object`

Parent class for Model and Parallel_Link_Model subclasses; holds common defs. This cannot be used to instantiate a functioning model directly. Use a subclass FlexModel or PerformanceModel

add_demand(*source_node_name*, *dest_node_name*, *traffic=0*, *name='none'*)

Adds a traffic load (Demand) from point A to point B in the model and validates model.

Parameters

- **source_node_name** – name of Demand's source Node
- **dest_node_name** – name of Demand's destination Node
- **traffic** – amount of traffic (magnitude) of the Demand
- **name** – Demand name

Returns A validated Model object with the new demand

add_node(*node_object*)

Adds a node object to the model object and validates self

Parameters **node_object** – Node object to add to self

add_rsvp_lsp(*source_node_name*, *dest_node_name*, *name*)

Adds an RSVP LSP with name from the source node to the dest node and validates model.

Parameters

- **source_node_name** – LSP source Node name
- **dest_node_name** – LSP destination Node name
- **name** – name of LSP

Returns A validated Model with the new RSVP_LSP object

add_srlg (*srlg_name*)
Adds SRLG object to Model

Parameters **srlg_name** – name of SRLG to add to self

all_interface_circuit_ids
Returns all interface circuit_ids

change_interface_name (*node_name*, *current_interface_name*, *new_interface_name*)
Changes interface name

Parameters

- **node_name** – name of Node holding Interface
- **current_interface_name** – current Interface name
- **new_interface_name** – new Interface name

Returns Interface with new name

display_demand_paths ()
Displays each demand and its path(s) across the network

display_interface_objects ()
Displays interface objects in a more human readable manner

display_interface_status ()
Returns failed = True/False for each interface

display_interfaces_traffic ()
A human-readable(-ish) display of interfaces and traffic on each

display_node_status ()
Returns failed = True/False for each node

fail_interface (*interface_name*, *node_name*)
Fails the Interface in self object for the interface_name/node_name pair

Parameters

- **interface_name** – name of Interface object
- **node_name** – Name of Node holding Interface

fail_node (*node_name*)
Fails specified Node with name node_name

fail_srlg (*srlg_name*)
Sets SRLG with name srlg_name to failed = True

Parameters **srlg_name** – name of SRLG to fail

Returns none

get_circuit_object_from_interface (*interface_name*, *node_name*)
Returns a Circuit object, given a Node name and Interface name

Parameters

- **interface_name** – Interface object on one side of Circuit
- **node_name** – Node name where Interface resides

Returns Circuit object from self that contains Interface with interface_name and node_name

get_demand_object(*source_node_name*, *dest_node_name*, *demand_name='none'*)
Returns demand specified by the source_node_name, dest_node_name, name; throws exception if demand not found

Parameters

- **source_node_name** – name of Node where desired Demand originates (source)
- **dest_node_name** – name of Node where desired Demand terminates (destination)
- **demand_name** – name of Demand object

Returns desired Demand object that matches parameters above

get_demand_objects_dest_node(*dest_node_name*)
Returns list of demands objects originating at the destination node

Parameters **dest_node_name** – name of destination node for Demands

Returns list of Demands terminating on destination node

get_demand_objects_source_node(*source_node_name*)
Returns list of demand objects originating at the node with name source_node_name

Parameters **source_node_name** – name of source node for Demands

Returns list of Demands originating at node

get_failed_interface_objects()
Returns a list of all failed interfaces in self

get_failed_node_objects()
Returns a list of all failed Nodes in self

get_interface_object(*interface_name*, *node_name*)
Returns an interface object for specified node name and interface name

Parameters

- **interface_name** – name of Interface
- **node_name** – name of Node

Returns Specified Interface object from self

get_node_interfaces(*node_name*)
Returns list of interfaces on specified node name

get_node_object(*node_name*)
Returns a Node object from self, given a Node's name

Parameters **node_name** – name of Node object in self

Returns Node object with node_name

get_non_failed_node_objects()
Returns a list of all failed nodes

get_orphan_node_objects()
Returns list of Nodes that have no interfaces

get_rsvp_lsp(*source_node_name*, *dest_node_name*, *lsp_name='none'*)
Returns the RSVP LSP from the model with the specified source node name, dest node name, and LSP name.

Parameters

- **`source_node_name`** – name of source node for LSP
- **`dest_node_name`** – name of destination node for LSP
- **`lsp_name`** – name of LSP

Returns RSVP_LSP object

`get_srlg_object`(*srlg_name*, *raise_exception=True*)
Returns SRLG in self with srlg_name

Parameters

- **`srlg_name`** – name of SRLG
- **`raise_exception`** – raise an exception if SRLG with name=srlg_name does not exist in self

Returns None

`get_unfailed_interface_objects`()
Returns a list of all non-failed interfaces in the self

`get_unrouted_demand_objects`()
Returns list of demand objects that cannot be routed in self

`is_node_an_orphan`(*node_object*)
Determines if a node is in orphan_nodes. A node in orphan_nodes is a Node with no Interface objects

Parameters **`node_object`** – Node object

Returns Boolean indicating if Node is orphan (True) or not (False)

`parallel_demand_groups`()
Determine demands with same source and dest nodes

Returns dict with entries where key is 'source_node_name-dest_node_name' and value is a list of demands with matching source/dest nodes # noqa E501

Example:

```
{'A-F': [Demand(source = A, dest = F, traffic = 40, name = 'dmd_a_f_1')],
'A-D': [Demand(source = A, dest = D, traffic = 80, name = 'dmd_a_d_1'),
Demand(source = A, dest = D, traffic = 70, name = 'dmd_a_d_2'),
Demand(source = A, dest = D, traffic = 100, name = 'dmd_a_to_d_3')],
'F-E': [Demand(source = F, dest = E, traffic = 400, name = 'dmd_f_e_1')]}
```

`parallel_lsp_groups`()
Determine LSPs with same source and dest nodes

Returns dict with entries where key is 'source_node_name-dest_node_name' and value is a list of LSPs with matching source/dest nodes # noqa E501

Example:

```
{'A-F': [RSVP_LSP(source = A, dest = F, lsp_name = 'lsp_a_f')],
'A-D': [RSVP_LSP(source = A, dest = D, lsp_name = 'lsp_a_d_1'),
RSVP_LSP(source = A, dest = D, lsp_name = 'lsp_a_d_2'),
RSVP_LSP(source = A, dest = D, lsp_name = 'lsp_a_d_4'),
RSVP_LSP(source = A, dest = D, lsp_name = 'lsp_a_d_3')],
'B-C': [RSVP_LSP(source = B, dest = C, lsp_name = 'lsp_b_c_1')],
'F-E': [RSVP_LSP(source = F, dest = E, lsp_name = 'lsp_f_e_1')]}
```

simulation_diagnostics()

Analyzes simulation results and looks for the following:

- Number of routed LSPs carrying Demands
- Number of routed LSPs with no Demands
- Number of Demands riding LSPs
- Number of Demands not riding LSPs
- Number of unrouted LSPs
- Number of unrouted Demands

Returns dict with the above as keys and the quantity of each for values and generators for routed LSPs with no Demands, routed LSPs carrying Demands, Demands riding LSPs # noqa E501

This is not cached currently and my be expensive to (re)run on a very large model. Current best practice is to assign the output of this to a variable:

Example:

```
sim_diag1 = model1.simulation_diagnostics()
```

unfail_interface(*interface_name*, *node_name*, *raise_exception=False*)

Unfails the Interface object for the interface_name, node_name pair.

Parameters

- **interface_name** – name of interface
- **node_name** – node name
- **raise_exception** – If raise_exception=True, an exception will be raised if the interface cannot be unfailed. An example of this would be if you tried to unfail the interface when the parent node or remote node was in a failed state

Returns Interface object from Model with 'failed' attribute set to False

unfail_node(*node_name*)

Unfails the Node with name=node_name

unfail_srlg(*srlg_name*)

Sets SRLG with srlg_name to failed = False :param srlg_name: name of SRLG to unfail :return: none

4.12.2 PerformanceModel

class pyNTM.performance_model.**PerformanceModel**(*interface_objects={}*, *node_objects={}*, *demand_objects={}*, *rsvp_lsp_objects={}*)

Bases: *pyNTM.master_model._MasterModel*

A network model object consisting of the following base components:

- Interface objects (set): layer 3 Node interfaces. Interfaces have a 'capacity' attribute that determines how much traffic it can carry. Note: Interfaces are matched into Circuit objects based on the interface circuit_ids –> A pair of Interfaces with the same circuit_id value get matched into a Circuit
- Node objects (set): vertices on the network (aka 'layer 3 devices') that contain Interface objects. Nodes are connected to each other via a pair of matched Interfaces (Circuits)

- Demand objects (set): traffic loads on the network. Each demand starts from a source node and transits the network to a destination node. A demand also has a magnitude, representing how much traffic it is carrying. The demand's magnitude will apply against each interface's available capacity
- RSVP LSP objects (set): RSVP LSPs in the Model
- Circuit objects are created by matching Interface objects

add_circuit(*node_a_object*, *node_b_object*, *node_a_interface_name*, *node_b_interface_name*, *cost_intf_a=1*, *cost_intf_b=1*, *capacity=1000*, *failed=False*, *circuit_id=None*)

Creates component Interface objects for a new Circuit in the Model. The Circuit object will then be created during the validate_model() call.

Parameters

- **node_a_object** – Node object
- **node_b_object** – Node object
- **node_a_interface_name** – name of component Interface on node_a
- **node_b_interface_name** – name of component Interface on node_b
- **cost_intf_a** – metric/cost of node_a_interface component Interface
- **cost_intf_b** – metric/cost of node_b_interface component Interface
- **capacity** – Circuit's capacity
- **failed** – Should the Circuit be created in a Failed state?
- **circuit_id** – Optional. Will be auto-assigned unless specified

Returns Model with new Circuit comprised of 2 new Interfaces

add_network_interfaces_from_list(*network_interfaces*)

A tool that reads network interface info and updates an *existing* model. Intended to be used from CLI/interactive environment Interface info must be a list of dicts and in format like below example.

Example:

```
network_interfaces = [
{'name':'A-to-B', 'cost':4,'capacity':100, 'node':'A',
'remote_node': 'B', 'circuit_id': 1, 'failed': False},
{'name':'A-to-Bv2', 'cost':40,'capacity':150, 'node':'A',
'remote_node': 'B', 'circuit_id': 2, 'failed': False},
{'name':'A-to-C', 'cost':1,'capacity':200, 'node':'A',
'remote_node': 'C', 'circuit_id': 3, 'failed': False},]
```

Parameters **network_interfaces** – python list of attributes for Interface objects

Returns self with new Interface objects

get_all_paths_reservable_bw(*source_node_name*, *dest_node_name*, *include_failed_circuits=True*, *cutoff=10*, *needed_bw=0*)

For a source and dest node name pair, find all simple path(s) with at least needed_bw reservable bandwidth available less than or equal to cutoff hops long. The amount of simple paths (paths that don't have repeating nodes) can be very large for larger topologies and so this call can be very expensive. Use the cutoff argument to limit the path length to consider to cut down on the time it takes to run this call.

Parameters

- **source_node_name** – name of source node in path
- **dest_node_name** – name of destination node in path

- **include_failed_circuits** – include failed circuits in the topology
- **needed_bw** – the amount of reservable bandwidth required on the path
- **cutoff** – max amount of path hops

Returns Return the path(s) in dictionary form:

Example:

```
>>> model.get_all_paths_reservable_bw('A', 'B', False, 5, 10)
{'path': [
[Interface(name = 'A-to-D', cost = 40, capacity = 20.0,
  node_object = Node('A'), remote_node_object = Node('D'), circuit_id = 2),
Interface(name = 'D-to-B', cost = 20, capacity = 125.0, node_object = Node('D
↪'),
  remote_node_object = Node('B'), circuit_id = 7)],
[Interface(name = 'A-to-D', cost = 40, capacity = 20.0, node_object = Node('A
↪'),
  remote_node_object = Node('D'), circuit_id = 2),
Interface(name = 'D-to-G', cost = 10, capacity = 100.0, node_object = Node('D
↪'),
  remote_node_object = Node('G'), circuit_id = 8),
Interface(name = 'G-to-B', cost = 10, capacity = 100.0, node_object = Node('G
↪'),
  remote_node_object = Node('B'), circuit_id = 9)]
]}
```

get_interface_object_from_nodes(*local_node_name*, *remote_node_name*)
Returns a list of Interface objects with the specified local and remote node names.

Parameters

- **local_node_name** – Name of Interface local node
- **remote_node_name** – Name of Interface remote node

Returns Interface object with specified local node and remote node names

get_shortest_path(*source_node_name*, *dest_node_name*, *needed_bw=0*)
For a source and dest node name pair, find the shortest path(s) with at least needed_bw available.

Parameters

- **source_node_name** – name of source node in path
- **dest_node_name** – name of destination node in path
- **needed_bw** – the amount of reservable bandwidth required on the path

Returns

Return the shortest path in dictionary form

Example:

```
shortest_path = {'path': [list of shortest path routes], 'cost':␣
↪path_cost}
```

get_shortest_path_for_routed_lsp(*source_node_name*, *dest_node_name*, *lsp*, *needed_bw*)
For a source and dest node name pair, find the shortest path(s) with at least needed_bw available for an LSP that is already routed. This takes into account reserved bandwidth on the Interfaces the LSP already transits, allowing the bandwidth reserved by the LSP to be considered for reservation on any new path for

the input LSP Return the shortest path in dictionary form: shortest_path = {'path': [list of shortest path routes], 'cost': path_cost}

Parameters

- **source_node_name** – name of source node
- **dest_node_name** – name of destination node
- **lsp** – LSP object
- **needed_bw** – reserved bandwidth for LSPs

Returns dict {'path': [list of lists, each list a shortest path route], 'cost': path_cost}

Example::

```
>>> lsp
RSVP_LSP(source = B, dest = C, lsp_name = 'lsp_b_c_1')
>>> path = model.get_shortest_path_for_routed_lsp('A', 'D', lsp, 10)
>>> path
{'path': [[Interface(name = 'A-to-D', cost = 40, capacity = 20.0, node_
↪object = Node('A'),
remote_node_object = Node('D'), circuit_id = 3)]], 'cost': 40}
```

classmethod load_model_file(*data_file*)

Opens a network_modeling data file, returns a model containing the info in the data file, and runs update_simulation().

The data file must be of the appropriate format to produce a valid model. This cannot be used to open multiple models in a single python instance - there may be unpredictable results in the info in the models. The format for the file must be a tab separated value file. This docstring you are reading may not display the table info explanations/examples below correctly on https://pyntm.readthedocs.io/en/latest/api.html. Recommend either using help(Model.load_model_file) at the python3 cli or looking at one of the sample model data_files in github: https://github.com/tim-fiola/network_traffic_modeler_py3/blob/master/examples/sample_network_model_file.csv https://github.com/tim-fiola/network_traffic_modeler_py3/blob/master/examples/lsp_model_test_file.csv The following headers must exist, with the following tab-column names beneath:

```
INTERFACES_TABLE
- node_object_name - name of node   where interface resides
- remote_node_object_name   - name of remote node
- name - interface name
- cost - IGP cost/metric for interface
- capacity - capacity
- rsvp_enabled (optional) - is interface allowed to carry RSVP LSPs?␣
↪True|False; default is True
- percent_reservable_bandwidth (optional) - percent of capacity allowed to be␣
↪reserved by RSVP LSPs; this
value should be given as a percentage value - ie 80% would be given as 80,␣
↪NOT .80.  Default is 100
```

Note - The existence of Nodes will be inferred from the INTERFACES_TABLE. So a Node created from an Interface does not have to appear in the NODES_TABLE unless you want to add additional attributes for the Node such as latitude/longitude

NODES_TABLE - - name - name of node - lon - longitude (or y-coordinate) (optional) - lat - latitude (or x-coordinate) (optional)

Note - The NODES_TABLE is present for 2 reasons: - to add a Node that has no interfaces - and/or to add additional attributes for a Node inferred from the INTERFACES_TABLE

DEMANDS_TABLE - source - source node name - dest - destination node name - traffic - amount of traffic on demand - name - name of demand

RSVP_LSP_TABLE - source - LSP's source node - dest - LSP's destination node - name - name of LSP - configured_setup_bw (optional) - if LSP has a fixed, static configured setup bandwidth, place that static value here, if LSP is auto-bandwidth, then leave this blank for the LSP - manual_metric (optional) - manually assigned metric for LSP, if not using default metric from topology shortest path

Functional model files can be found in this directory in https://github.com/tim-fiola/network_traffic_modeler_py3/tree/master/examples Here is an example of a data file:

Example:

```
INTERFACES_TABLE
node_object_name    remote_node_object_name name    cost    capacity    rsvp_
↪enabled    percent_reservable_bandwidth  # noqa E501
A   B       A-to-B  4       100
B   A       B-to-A  4       100

NODES_TABLE
name        lon     lat
A   50      0
B   0       -50

DEMANDS_TABLE
source      dest    traffic name
A   B       80      dmd_a_b_1

RSVP_LSP_TABLE
source      dest    name    configured_setup_bw manual_metric
A   B       lsp_a_b_1   10  15
A   B       lsp_a_b_2       10
```

Parameters **data_file** – file with model info

Returns Model object

update_simulation()
Updates the simulation state; this needs to be run any time there is a change to the state of the Model, such as failing an interface, adding a Demand, adding/removing and LSP, etc. This call does not carry forward any state from the previous simulation results.

validate_model()
Validates that data fed into the model creates a valid network model

validate_srlg_nodes()
Validate that Nodes in each SRLG have the SRLG in their srlgs set. srlg_errors is a dict of node names as keys and a list of SRLGs that node is a member of in the model but that the SRLG is not in node.srlgs :return: dict where keys are Node names and values are lists of SRLG names; each value will be a single list of SRLG names missing that Node in the SRLG node set

4.12.3 FlexModel

class pyNTM.flex_model.**FlexModel**(*interface_objects={}*, *node_objects={}*, *demand_objects={}*, *rsvp_lsp_objects={}*)

Bases: *pyNTM.master_model._MasterModel*

A network model object consisting of the following base components:

- Interface objects (set): layer 3 Node interfaces. Interfaces have a 'capacity' attribute that determines how much traffic it can carry. Note: Interfaces are matched into Circuit objects based on the interface circuit_ids –> A pair of Interfaces with the same circuit_id value get matched into a Circuit
- Node objects (set): vertices on the network (aka 'layer 3 devices') that contain Interface objects. Nodes are connected to each other via a pair of matched Interfaces (Circuits)
- Demand objects (set): traffic loads on the network. Each demand starts from a source node and transits the network to a destination node. A demand also has a magnitude, representing how much traffic it is carrying. The demand's magnitude will apply against each interface's available capacity
- RSVP LSP objects (set): RSVP LSPs in the Model
- Circuit objects are created by matching Interface objects using common circuit_id

add_circuit(*node_a_object*, *node_b_object*, *node_a_interface_name*, *node_b_interface_name*, *cost_intf_a=1*, *cost_intf_b=1*, *capacity=1000*, *failed=False*, *circuit_id=None*)

Creates component Interface objects for a new Circuit in the Model. The Circuit object will then be created during the validate_model() call.

Parameters

- **node_a_object** – Node object
- **node_b_object** – Node object
- **node_a_interface_name** – name of component Interface on node_a
- **node_b_interface_name** – name of component Interface on node_b
- **cost_intf_a** – metric/cost of node_a_interface component Interface
- **cost_intf_b** – metric/cost of node_b_interface component Interface
- **capacity** – Circuit's capacity
- **failed** – Should the Circuit be created in a Failed state?
- **circuit_id** – Optional. Will be auto-assigned unless specified

Returns Model with new Circuit comprised of 2 new Interfaces

add_network_interfaces_from_list(*network_interfaces*)

A tool that reads network interface info and updates an *existing* model. Intended to be used from CLI/interactive environment Interface info must be a list of dicts and in format like below example.

Example:

```
network_interfaces = [
{'name':'A-to-B', 'cost':4,'capacity':100, 'node':'A',
'remote_node': 'B', 'circuit_id': 1, 'failed': False},
{'name':'A-to-Bv2', 'cost':40,'capacity':150, 'node':'A',
'remote_node': 'B', 'circuit_id': 2, 'failed': False},
{'name':'A-to-C', 'cost':1,'capacity':200, 'node':'A',
'remote_node': 'C', 'circuit_id': 3, 'failed': False},]
```

Parameters **network_interfaces** – python list of attributes for Interface objects

Returns self with new Interface objects

find_igp_shortcuts(*paths*, *node_paths*)

Check for LSPs along the shortest path; find the first LSP the demand can take with a source and destination that is on the LSP's IGP path

1. examine each IGP path
2. If none of the nodes on the path have IGP shortcuts, continue to next path
3. If some nodes have IGP shortcuts enabled, note the hop number (1, 2, 3, etc)
4. For nodes that have IGP shortcuts, is there an LSP from that node to a downstream node on the path?

- if yes, compare the IGP metric of the path to the LSP remote node to that of the LSP metric to that node
- if no, look at next node downstream with IGP shortcuts

5. Look for manually set RSVP LSP metrics that may alter the path calculations

Parameters

- **paths** – List of lists; each list contains egress Interfaces along the path from source to destination (ordered from source to destination) # noqa E501
- **node_paths** – List of lists; each list contains node names along the path from source to destination (ordered from source to destination)

Returns List of lists; each list contains Interfaces and/or RSVP LSPs along each path from source to destination # noqa E501

get_all_paths_reservable_bw(*source_node_name*, *dest_node_name*, *include_failed_circuits=True*, *cutoff=10*, *needed_bw=0*)

For a source and dest node name pair, find all simple path(s) with at least needed_bw reservable bandwidth available less than or equal to cutoff hops long.

The amount of simple paths (paths that don't have repeating nodes) can be very large for larger topologies and so this call can be very expensive. Use the cutoff argument to limit the path length to consider to cut down on the time it takes to run this call.

Parameters

- **source_node_name** – name of source node in path
- **dest_node_name** – name of destination node in path
- **include_failed_circuits** – include failed circuits in the topology
- **needed_bw** – the amount of reservable bandwidth required on the path
- **cutoff** – max amount of path hops

Returns Return the path(s) in dictionary form: path = {'path': [list of shortest path routes]}

Example:

```
>>> model.get_all_paths_reservable_bw('A', 'B', False, 5, 10)
{'path': [
[Interface(name = 'A-to-D', cost = 40, capacity = 20.0,
node_object = Node('A'), remote_node_object = Node('D'), circuit_id = 2),
Interface(name = 'D-to-B', cost = 20, capacity = 125.0, node_object = Node('D
↪'),
```

(continues on next page)

(continued from previous page)

```
remote_node_object = Node('B'), circuit_id = 7)],
[Interface(name = 'A-to-D', cost = 40, capacity = 20.0, node_object = Node('A
↪'),
remote_node_object = Node('D'), circuit_id = 2),
Interface(name = 'D-to-G', cost = 10, capacity = 100.0, node_object = Node('D
↪'),
remote_node_object = Node('G'), circuit_id = 8),
Interface(name = 'G-to-B', cost = 10, capacity = 100.0, node_object = Node('G
↪'),
remote_node_object = Node('B'), circuit_id = 9)]
]}
```

`get_interface_object_from_nodes`(*local_node_name*, *remote_node_name*, *circuit_id=None*)

Returns a list of Interface objects with the specified local and remote node names.

If 'circuit_id' is not specified, may return a list of len > 1, as multiple/parallel interfaces are allowed in Parallel_Link_Model objects.

If 'circuit_id' is specified, will return a list of len == 1, as specifying the 'circuit_id' will narrow down any list of multiple interfaces to a single interface because circuit_ids bond interfaces on different nodes into a Circuit object.

Parameters

- **`local_node_name`** – Name of local node Interface resides on
- **`remote_node_name`** – Name of Interface's remote Node
- **`circuit_id`** – circuit_id of Interface (optional)

Returns list of Interface objects with common local node and remote node

`get_shortest_path`(*source_node_name*, *dest_node_name*, *needed_bw=0*)

For a source and dest node name pair, find the shortest path(s) with at least needed_bw available.

Parameters

- **`source_node_name`** – name of source node in path
- **`dest_node_name`** – name of destination node in path
- **`needed_bw`** – the amount of reservable bandwidth required on the path

Returns Return the shortest path in dictionary form: shortest_path = {'path': [list of shortest path routes], 'cost': path_cost}

`get_shortest_path_for_routed_lsp`(*source_node_name*, *dest_node_name*, *lsp*, *needed_bw*)

For a source and dest node name pair, find the shortest path(s) with at least needed_bw available for an LSP that is already routed. Return the shortest path in dictionary form: shortest_path = {'path': [list of shortest path routes], 'cost': path_cost}

Parameters

- **`source_node_name`** – name of source node
- **`dest_node_name`** – name of destination node
- **`lsp`** – LSP object
- **`needed_bw`** – reserved bandwidth for LSPs

Returns dict {'path': [list of lists, each list a shortest path route], 'cost': path_cost}

classmethod load_model_file(*data_file*)

Opens a network_modeling data file, returns a model containing the info in the data file, and runs update_simulation().

The data file must be of the appropriate format to produce a valid model. This cannot be used to open multiple models in a single python instance - there may be unpredictable results in the info in the models.

The format for the file must be a tab separated value file.

CIRCUIT ID (circuit_id) MUST BE SPECIFIED AS THIS IS WHAT ALLOWS THE CLASS TO DISCERN WHAT MULTIPLE, PARALLEL INTERFACES BETWEEN THE SAME NODES MATCH UP INTO WHICH CIRCUIT. THE circuit_id CAN BE ANY COMMON KEY, SUCH AS IP SUBNET ID OR DESIGNATED CIRCUIT ID FROM PRODUCTION.

This docstring you are reading may not display the table info explanations/examples below correctly on https://pyntm.readthedocs.io/en/latest/api.html. Recommend either using help(Model.load_model_file) at the python3 cli or looking at one of the sample model data_files in github: https://github.com/tim-fiola/network_traffic_modeler_py3/blob/master/examples/sample_network_model_file.csv https://github.com/tim-fiola/network_traffic_modeler_py3/blob/master/examples/lsp_model_test_file.csv

The following headers must exist, with the following tab-column names beneath:

```
INTERFACES_TABLE
- node_object_name - name of node   where interface resides
- remote_node_object_name   - name of remote node
- name - interface name
- cost - IGP cost/metric for interface
- capacity - capacity
- circuit_id - id of the circuit; used to match two Interfaces into Circuits;
    - each circuit_id can only appear twice in the model
    - circuit_id can be string or integer
- rsvp_enabled (optional) - is interface allowed to carry RSVP LSPs?␣
↪True|False; default is True
- percent_reservable_bandwidth (optional) - percent of capacity allowed to be␣
↪reserved by RSVP LSPs; this
value should be given as a percentage value - ie 80% would be given as 80,␣
↪NOT .80.  Default is 100

Note - The existence of Nodes will be inferred from the INTERFACES_TABLE.
So a Node created from an Interface does not have to appear in the
NODES_TABLE unless you want to add additional attributes for the Node
such as latitude/longitude

NODES_TABLE -
- name - name of node
- lon - longitude (or y-coordinate)
- lat - latitude (or x-coordinate)
- igp_shortcuts_enabled(default=False)

Note - The NODES_TABLE is present for 2 reasons:
- to add a Node that has no interfaces
- and/or to add additional attributes for a Node inferred from
the INTERFACES_TABLE

DEMANDS_TABLE
- source - source node name
- dest - destination node name
- traffic   - amount of traffic on demand
- name - name of demand
```

(continues on next page)

(continued from previous page)

```
RSVP_LSP_TABLE (this table is optional)
- source - source node name
- dest - destination node name
- name - name of LSP
- configured_setup_bw - if LSP has a fixed, static configured setup bandwidth,
↪ place that static value here,
if LSP is auto-bandwidth, then leave this blank for the LSP (optional)
- manual_metric - manually assigned metric for LSP, if not using default␣
↪metric from topology
shortest path (optional)
```

Functional model files can be found in this directory in https://github.com/tim-fiola/network_traffic_modeler_py3/tree/master/examples

Here is an example of a data file.

Example:

```
INTERFACES_TABLE
node_object_name    remote_node_object_name name    cost    capacity    ␣
↪circuit_id  rsvp_enabled    percent_reservable_bandwidth # noqa E501
A   B       A-to-B_1    20  120 1   True  50
B   A       B-to-A_1    20  120 1   True  50
A   B   A-to-B_2    20  150 2
B   A   B-to-A_2    20  150 2
A   B   A-to-B_3    10  200 3   False
B   A   B-to-A_3    10  200 3   False

NODES_TABLE
name        lon     lat igp_shortcuts_enabled(default=False)
A   50      0    True
B   0       -50 False

DEMANDS_TABLE
source      dest    traffic name
A   B       80      dmd_a_b_1

RSVP_LSP_TABLE
source      dest    name    configured_setup_bw manual_metric
A   B       lsp_a_b_1   10  19
A   B       lsp_a_b_2       6
```

Parameters **data_file** – file with model info

Returns Model object

update_simulation()

Updates the simulation state; this needs to be run any time there is a change to the state of the Model, such as failing an interface, adding a Demand, adding/removing and LSP, etc.

This call does not carry forward any state from the previous simulation results.

validate_model()

Validates that data fed into the model creates a valid network model

4.12.4 Node

class `pyNTM.node.`**`Node`**(*name*, *lat=0*, *lon=0*)

Bases: `object`

A class to represent a layer 3 device in the model.

Attribute lat, lon can be used as y, x values, respectively for graphing purposes.

`add_to_srlg`(*srlg_name*, *model*, *create_if_not_present=False*)

Adds self to an SRLG with name=srlg_name in model.

Parameters

- **`srlg_name`** – name of srlg
- **`model`** – Model object
- **`create_if_not_present`** – Boolean. Create the SRLG if it does not exist in model already. True will create SRLG in model; False will raise ModelException # noqa E501

Returns None

`adjacent_nodes`(*model*)

Returns a list of adjacent nodes

Parameters **`model`** – model Object

Returns List of adjacent Nodes in model

`failed`

Is node failed? Boolean. It is NOT recommended to directly modify this property. Rather, Model methods fail_node(node_name) and unfail_node(node_name)

Returns Boolean - is node failed?

`igp_shortcuts_enabled`

Are IGP shortcuts enabled for RSVP LSPs on this Node? This is only applicable in the FlexModel; PerformanceModel subclass ignores this attribute

`interfaces`(*model*)

Returns interfaces for a given node

Parameters **`model`** – model structure

Return adjacency_list (list) list of interfaces on the given node

`lat`

Latitude or y-coordinate of Node on a plot

`lon`

Longitude or x-coordinate of Node on a plot

`remove_from_srlg`(*srlg_name*, *model*)

Removes self from SRLG with srlg_name in model

Parameters

- **`srlg_name`** – name of SRLG
- **`model`** – Model object

Returns none

`srlgs`

4.12.5 Interface

class `pyNTM.interface.`**`Interface`**(*name*, *cost*, *capacity*, *node_object*, *remote_node_object*, *circuit_id=None*, *rsvp_enabled=True*, *percent_reservable_bandwidth=100*)

Bases: `object`

An object representing a Node's Interface

`add_to_srlg`(*srlg_name*, *model*, *create_if_not_present=False*)

Adds self to an SRLG with name=srlg_name in model. Also finds the remote Interface object and adds that to the SRLG.

Parameters

- **`srlg_name`** – name of srlg
- **`model`** – Model object
- **`create_if_not_present`** – Boolean. Create the SRLG if it does not exist in model already. True will create SRLG in model; False will raise ModelException # noqa E501

Returns None

`capacity`

`cost`

`demands`(*model*)

Returns list of demands that egress the interface

Parameters **`model`** – model object containing self

Returns list of Demand objects egressing self

`fail_interface`(*model*)

Updates the specified interface and the remote interface with failed==True

Parameters **`model`** – model object containing self

`failed`

Is interface failed? Boolean. It is NOT recommended to directly modify this property. Rather, use Interface.fail or Interface.unfail.

Returns Boolean - is Interface failed?

`get_circuit_object`(*model*)

Returns the circuit object from the model that an interface is associated with.

Parameters **`model`** – model object containing self

Returns Circuit object containing self

`get_remote_interface`(*model*)

Returns Interface on other side of the Circuit containing self

Parameters **`model`** – model object holding self

Returns Interface object on remote side of Circuit containing self

`lsps`(*model*)

Returns a list of RSVP LSPs that egress the interface

Parameters **`model`** – Model object

Returns list of RSVP LSPs that egress the interface

remove_from_srlg(*srlg_name*, *model*)

Removes self and remote interface object from SRLG with srlg_name in model.

Parameters

- **srlg_name** – name of SRLG
- **model** – Model object

Returns none

reservable_bandwidth

Amount of bandwidth available for rsvp lsp reservation. If interface is not rsvp_enabled, then reservable_bandwidth is set to -1

reserved_bandwidth

Amount of interface capacity reserved by RSVP LSPs

srlgs

unfail_interface(*model*)

Updates the specified interface and the remote interface with failed==False

Parameters model – model object containing self

utilization

Returns utilization percent = (self.traffic/self.capacity)*100

4.12.6 Demand

class `pyNTM.demand.`**Demand**(*source_node_object*, *dest_node_object*, *traffic=0*, *name='none'*)

Bases: `object`

A representation of traffic load on the modeled network

path_detail

Returns a detailed breakdown of the Demand path. Each path will have the following information:

items: The combination of Interfaces and/or LSPs that the Demand takes from source to destination

splits: each item on the path (Interface and/or LSP) and the number of cumulative ECMP path splits that the Demand has transited as it egresses the source node for that element.

Splits can be used to calculate how much of the Demand's traffic is on a certain path (see path_traffic below) or how much of the Demand's traffic is on a certain element.

The demand object's `path_detail` property can be very useful to determine how much of the demand's traffic egresses each object (interface, LSP) in the path.

For example, sample demand `Demand(source = A, dest = E, traffic = 24, name = 'dmd_a_e_1')` has 24 units of traffic.

Here is the `path_0` entry for the sample demand:

```
'path_0': {
    'items': [Interface(name = 'A-to-B', cost = 4, capacity = 100, node_
↪object = Node('A'),
                remote_node_object = Node('B'), circuit_id = '1'),
            Interface(name = 'B-to-E_3', cost = 3, capacity = 200, node_
↪object = Node('B'),
                remote_node_object = Node('E'), circuit_id = '27')
    ],
```

(continues on next page)

(continued from previous page)

```
    'path_traffic': 4.0,
    'splits': {Interface(name = 'A-to-B', cost = 4, capacity = 100, node_
↪object = Node('A'),
                remote_node_object = Node('B'), circuit_id = '1'): 2,
              Interface(name = 'B-to-E_3', cost = 3, capacity = 200, node_
↪object = Node('B'),
                remote_node_object = Node('E'), circuit_id = '27'): 6}
}
```

The `path_0` component of the `path_detail` property in this example shows the following:

- `Interface(name = 'A-to-B', cost = 4, capacity = 100, node_object = Node('A'), remote_node_object = Node('B'), circuit_id = '1')` has **2** splits
- `Interface(name = 'B-to-E_3', cost = 3, capacity = 200, node_object = Node('B'), remote_node_object = Node('E'), circuit_id = '27')` has **6** splits

To get the amount of traffic load from the specific demand that transits each interface, divide the amount of traffic that the demand has by the number of splits for the object:

- `Interface(name = 'A-to-B', cost = 4, capacity = 100, node_object = Node('A'), remote_node_object = Node('B'), circuit_id = '1')` carries **24 / 2 = 12** units of traffic from the sample demand.
- `Interface(name = 'B-to-E_3', cost = 3, capacity = 200, node_object = Node('B'), remote_node_object = Node('E'), circuit_id = '27')` carries **24 / 6 = 4** units of traffic from the sample demand.

Since the minimum amount of traffic found on any object in `path_0` is 4 units of traffic, `path_traffic` for `path_0` = 4.

Returns Dict of path entries (keys). The value for each key is another dict with 3 keys: 'items', 'splits', and 'path_traffic'. Each is described above. # noqa E501

4.12.7 Circuit

class `pyNTM.circuit.`**`Circuit`**(*interface_a*, *interface_b*)

Bases: `object`

A circuit is an object consisting of 2 connected interfaces

`circuit_id`()

Returns the circuit_id, which bonds the two component Interfaces to the Circuit

`failed`(*model*)

Is Circuit failed?

Parameters **`model`** – Model containing circuit

Returns Boolean

`get_circuit_interfaces`(*model*)

Return the Circuit's component Interface objects in model object

Parameters **`model`** – model object containing Circuit

Returns Component Interfaces in Circuit

4.12.8 RSVP_LSP

class pyNTM.rsvp.**RSVP_LSP**(*source_node_object*, *dest_node_object*, *lsp_name='none'*, *configured_setup_bandwidth=None*, *configured_manual_metric=None*)

Bases: `object`

A class to represent an RSVP label-switched-path in the network model

source_node_object: Node where LSP ingresses the network (LSP starts here)

dest_node_object: Node where LSP egresses the network (LSP ends here)

lsp_name: name of LSP

path:

```
will either be 'Unrouted' or be a dict containing the following -
- interfaces: list of interfaces that LSP egresses in the order it
    egresses them
- path_cost: sum of costs of the interfaces
- baseline_path_reservable_bw: the amount of reservable bandwidth
    available on the LSP's path when the LSP was signaled, not inclusive
    of the bandwidth already reserved by this LSP on that path (if any)
```

reserved_bandwidth: amount of bandwidth reserved by this LSP

setup_bandwidth: amount of bandwidth this LSP attempts to signal for when it sets up

demands_on_lsp(*model*)

Returns demands in model object that LSP is transporting.

Parameters **model** – model object containing LSP

Returns List of demands in model object that LSP carries

effective_metric(*model*)

Returns the manually assigned manual_metric (if defined) or the metric for the best path. The best path value will be the metric for the shortest possible path from LSP's source to dest, regardless of whether the LSP takes that shortest path or not.

Parameters **model** – model object containing self

Returns metric for the LSP's shortest possible path

find_rsvp_path_w_bw(*requested_bandwidth*, *model*)

Will search the topology of 'model' for a path for self that has at least 'requested_bandwidth' of reservable_bandwidth. If there is one, will update self.path; if not, will keep same self.path. When checking paths, this def will take into account its own reserved bandwidth if it is looking at paths that have interfaces already in its path['interfaces'] list.

Parameters

- **model** – Model object to search; this will typically be a Model object consisting of only non-failed interfaces # noqa E501
- **requested_bandwidth** – number of units set for reserved_bandwidth

Returns self with the current or updated path info

manual_metric

Manual metric for LSP. If set, this value will override the default (shortest path) metric for effective_metric.

This value must be a positive integer.

To restore the LSP's default metric (that of the shortest path) in a live simulation, set this value to -1.

`setup_bandwidth`
The bandwidth the LSP attempts to signal for.

Returns the bandwidth the LSP attempts to signal for

`topology_metric`(*model*)
Returns the metric sum of the interfaces that the LSP actually transits on the topology.

Parameters **`model`** – model object containing self

Returns sum of the metrics of the Interfaces that the LSP transits

`traffic_on_lsp`(*model*)
Returns the amount of traffic on the LSP

Parameters **`model`** – Model object for LSP

Returns Units of traffic on the LSP

4.12.9 Exceptions

Exceptions

`exception` `pyNTM.exceptions.`**`ModelException`**
Bases: `Exception`

4.13 Changelog

4.13.1 3.4.1

- Updated test environment to Focal linux (from Xenial) to allow `dash` and `dash-cytoscape` package import in CI/CD for visualization
  - Allows for moving WeatherMap from beta to production in future
  - It's no longer necessary to explicitly install the visualization requirements separately
- Heavy updates for docs
- Implemented black code formatting for local commits and in Travis CI/CD pipeline
- Implemented standard column name for RSVP LSP attribute to describe a manually assigned metric:
  - FlexModel had `lsp_metric` column name in docstrings and examples
  - PerformanceModel had `manual_metric` column name in docstrings and examples
  - Standardized column name to `manual_metric`
  - This was a purely cosmetic change as the actual `manual_metric` is based on the column's order in the table, not the specific column name

4.13.2 3.4.0

- This build, while functional, was yanked from pypi for reasons related to troubleshooting a new CI/CD pipeline. Let's just say there was some drama around that, and the migration to travis-ci.com. These things happen.
- 3.4.0 features moved to 3.4.1 build

4.13.3 3.3.1

- Fixed bug in FlexModel to account for complex topology scenario involving ECMP demand paths with multipe IGP shortcut LSPs and parallel links
- Added spacing_factor as a WeatherMap configurable parameter
- Added another test to functional tests within test_parallel_link_model for complex topology

4.13.4 3.3

- Import of the WeatherMap class must be done from pyNTM.weathermap instead of directly from pyNTM. This prevents a warning message that is otherwise superficial unless you are using the WeatherMap class

4.13.5 3.2

- Fixed bug in WeatherMap class that caused scripts with WeatherMap to run 2x and to not be able to run a WeatherMap class live from the python3 CLI

4.13.6 3.1

- Removed automatic call of load_model_file class methods performing update_simulation() call automatically.

4.13.7 3.0

- WeatherMap class added as beta feature to provide interactive visualization of network topology. This is a beta feature and is not undergoing unit testing. This feature is supported in the python3 interpreter, but not in the pypy3 interpreter. This feature gives a very interactive and informative visualization of the topology
- path_detail Demand property support in all Model classes; provides clarity on how much traffic is passing on a given Demand's path and how much of that traffic transits each component in the path
- Python 3.5 no longer supported
- Python 3.8 support added to unit/functional testing
- load_model_file class methods now perform update_simulation() call automatically. The update_simulation() call is only necessary to run after making a change to the topology after the model file has been loaded

4.13.8 2.1

- Enforcing tab-separated data in model data files (used to allow spaces or tabs between data entries)
- FlexModel class allows IGP RSVP shortcuts
- FlexModel and PerformanceModel classes allow/honor RSVP LSP manual metrics
- Made load_model_file for FlexModel and PerformanceModel classes more forgiving for number of lines allowed between tables

4.13.9 2.0

- Made version 1.7 into major version 2.0 to account for possible backwards compatibilty

4.13.10 1.7

- Renamed Model class to PerformanceModel
- Renamed Parallel_Link_Model class to FlexModel
- Optimization: general 18-25% performance improvement when measured by time to converge
- Moved common code from PerformanceModel and FlexModel to _MasterModel parent class
- Maintained unit testing coverage at 95%
- Cleaned up documentation/docstrings

4.13.11 1.6

- Added support for multiple links between nodes (Parallel_Link_Model)
- Cached parallel_lsp_groups in Model and Parallel_Link_Model objects (performance optimization)
- Added check for multiple links between nodes in Model object (not allowed)
- Added Parent Class _MasterModel to hold common defs for Model and Parallel_Link_Model subclasses
- Added simulation_diagnostics def in _MasterModel that gives potentially useful diagnostic info about the simulation results
- Simple user interface (beta feature) supports RSVP LSPs

4.13.12 1.5

- Updated code to account for networkx
- Improved some docstrings

4.13.13 1.4

- updated requirements.txt to allow use of beta features

4.13.14 1.3

- improved docstring for Model load_model_file class method
- updated requirements
- fixed bugs in beta features: visualization and simple UI
- updated unit testing

4.13.15 1.2

- added shared-risk link group (SRLG) support for Nodes and Interfaces
- added performance optimizations
- simplified sections of code

4.13.16 1.1

- added configured, fixed setup bandwidth capability on RSVP LSPs
- made small performance optimizations

4.13.17 1.0

- first release including pypi inetgration

4.13.18 previous releases

- versions prior to v1.0 were not released to pip, but distributed as a github directory
- initially a py2 version was made available here
- the py2 version is not maintained anymore in favor of the current py3 releases

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index