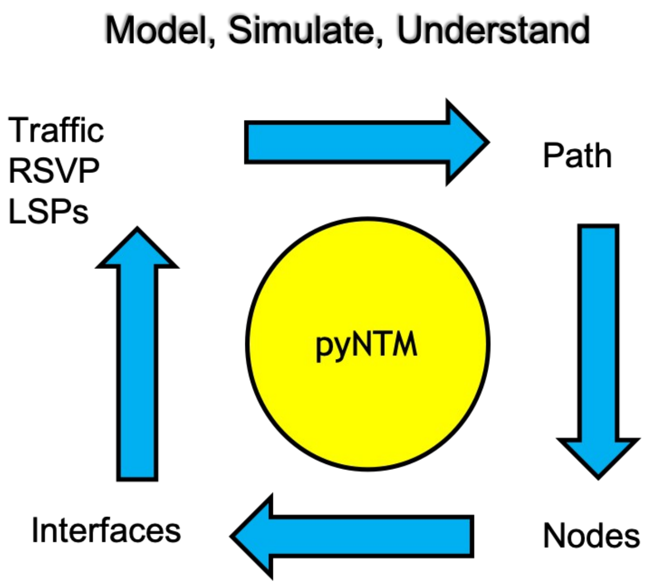
pyNTM Training Module 2 – getting started

Network Traffic Modeler in Python3

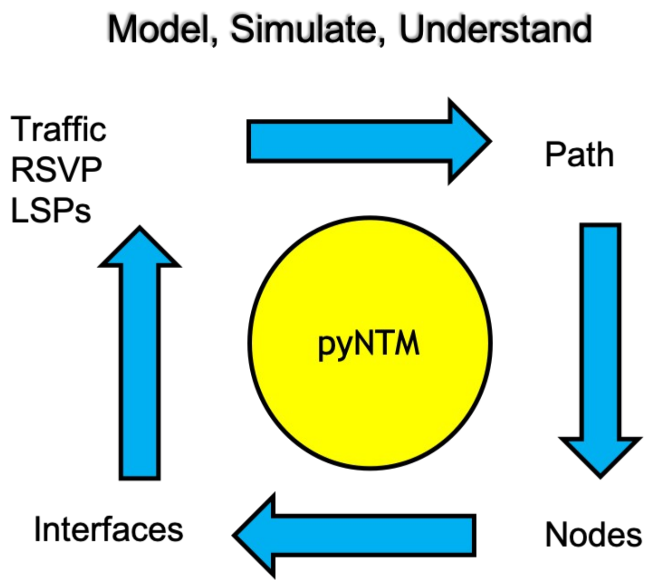
PyNTM version 3.3

Training version 2



Course topics

- Use python3!
- What is pyNTM?
- Get pyNTM
- Model Class Objects
- LIVE DEMO - setting up a virtual environment (if wanted/needed)
- About the Model file
- Setting up for the Exercises
- Converging the Model



(continued on next slide)

Course topics (continued)

- Live Exercise 1: shortest path(s) and Interface utilization
  - Examine IGP topology for the shortest path(s) between two Nodes
- Live Exercise 2: Failing an Interface
  - Failing an Interface and assessing impact on Interface utilization
- Live Exercise 3: Finding traffic Demands on an Interface
  - Determine which Demands are driving Interface utilization
- Live Exercise 4: Finding the path(s) of a specific Demand
  - Determine the ECMP path(s) a Demand takes through the IGP topology
- Live Exercise 5: Unfailing an Interface
- Interactive Visualization with the WeatherMap Class (beta)

What is pyNTM?

- pyNTM is the Network Traffic Modeler in python3
  - It is a modeling and simulation engine
  - It provides capability to define/modify a network topology
  - It provides capability to apply a traffic matrix to that topology to get simulation results
  - It does NOT provide a traffic matrix for your network
    - The user will have to create the traffic matrices for their network
    - Network modeling aside, in order to understand your network, it is imperative to understand the traffic matrix

Getting and using pyNTM

- Get pyNTM – 2 options
  - PyPI – PYthon Package Index
    - From OS CLI: *pip3 install pyNTM*
  - Download the repository from Github
    - https://github.com/tim-fiola/network_traffic_modeler_py3
    - pip3 installs current master branch
    - Dev branch has additional features under development
- Documentation
  - https://pyntm.readthedocs.io/en/latest/index.htm
  - Docstrings are also available via help call for the def

```
>>> help(Model.update_simulation)

Help on function update_simulation in module pyNTM.model:

update_simulation(self)
    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.
```

About the model file

- Tab separated file that describes
  - Interfaces
  - Nodes
    - Nodes can be inferred from their interface objects
    - The NODES_TABLE is present in the event that
      - You want to add other attributes for a Node (latitude, longitude)
      - You want to load an *orphan node* to your model (a Node that has no interfaces)
  - Demands
  - RSVP LSPs (if applicable)
- The example to the right shows a model data file for a PerformanceModel

INTERFACES_TABLE

node_object_name	remote_node_object_name	name	cost	capacity
A	B	A-to-B	4	100
A	C	A-to-C	1	200
A	D	A-to-D	8	150
B	A	B-to-A	4	100
B	D	B-to-D	7	200
B	E	B-to-E	3	200
D	B	D-to-B	7	200
D	C	D-to-C	9	150
D	A	D-to-A	8	150
D	E	D-to-E	4	100
D	F	D-to-F	3	100
C	A	C-to-A	1	200
C	D	C-to-D	9	150
E	B	E-to-B	3	200
E	D	E-to-D	4	100
F	D	F-to-D	3	100
F	B	F-to-B	6	100
B	F	B-to-F	6	100
G	H	G-to-H	4	150
H	G	H-to-G	4	150
G	D	G-to-D	2	50
D	G	D-to-G	2	50
H	E	H-to-E	4	100
E	H	E-to-H	4	100
E	F	E-to-F	3	100
F	E	F-to-E	3	100
H	D	H-to-D	4	100
D	H	D-to-H	4	100

NODES_TABLE

name	lon	lat
A	25	0
B	31	-177
C	60	-63
D	62	37
E	-60	124
F	2	35
G	30	90
H	52	124

DEMANDS_TABLE

source	dest	traffic	name
A	B	50	''
A	F	22	''
A	E	24	''
F	E	80	''
F	B	50	''
A	D	120	''
D	A	10	''
A	H	20	''
C	E	20	''
B	G	30	''
E	C	20	''

PerformanceModel files

- The docstrings for the *PerformanceModel.load_model_file* class method have more extensive explanations of how to format the PerformanceModel data file to accommodate the supported features
  - Also available at https://pyntm.readthedocs.io/en/latest/api.html#performancemodel

```
>>> help(PerformanceModel.load_model_file)
```

```
Help on method load_model_file in module pyNTM.performance_model:

load_model_file(data_file) method of builtins.type instance
    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
```

```
INTERFACES_TABLE
node_object_name    remote_node_object_name name    cost    capacity    rsvp_enabled    percent_reservable_bandwidth

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
```

About the model file (continued)

- The example to the right is a model file for a FlexModel
  - There is a required *circuit_id* entry in the INTERFACES_TABLE
    - Since the FlexModel allows for multiple Circuits between any 2 Nodes, a circuit_id must be specified for a deterministic match of the component Interfaces
  - There is an *igp_shortcuts_enabled* field in the NODES_TABLE
    - Specifies if IGP Shortcuts are enabled for the Node
      - Default is False
      - A True value allows Demands to use the LSPs on the Node even if the Demand source and destination don’t match the LSP’s source and destination

```
INTERFACES_TABLE
node_object_name	remote_node_object_name	name	cost	capacity	circuit_id
A	B	A-B	10	100	1
B	A	B-A	10	100	1
B	C	B-C	10	100	2
C	B	C-B	10	100	2
C	D	C-D	10	100	3
D	C	D-C	10	100	3
D	E	D-E	10	100	4
E	D	E-D	10	100	4
E	F	E-F	10	100	5
F	E	F-E	10	100	5
A	G	A-G	25	100	6
G	A	G-A	25	100	6
G	F	G-F	25	100	7
F	G	F-G	25	100	7


NODES_TABLE
name	lon	lat	igp_shortcuts_enabled
A	10	0	True
B	0	5	True
C	0	0	True
D	0	0	True
E	0	0	True
F	0	0	True


DEMANDS_TABLE
source	dest	traffic	name
A	F	10	dmd_a_f_1
D	F	8	dmd_d_f_1


RSVP_LSP_TABLE
source	dest	name
B	D	lsp_b_d_1
B	D	lsp_b_d_2
C	E	lsp_c_e_1
D	F	lsp_d_f_1
```

FlexModel files

```
>>> from pyNTM import FlexModel
>>> help(FlexModel.load_model_file)
```

- The docstrings for the Flex*Model.load_model_file* class method have more extensive explanations of how to format the FlexModel data file to accommodate the supported features
  - Also available at https://pyntm.readthedocs.io/en/latest/api.html#flexmodel

```
load_model_file(data_file) method of builtins.type instance
    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
        - manual_metric (optional) - manually assigned metric for LSP, if not using default metric from topology
        shortest path

        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)
```

```
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	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
```

About the Model Classes

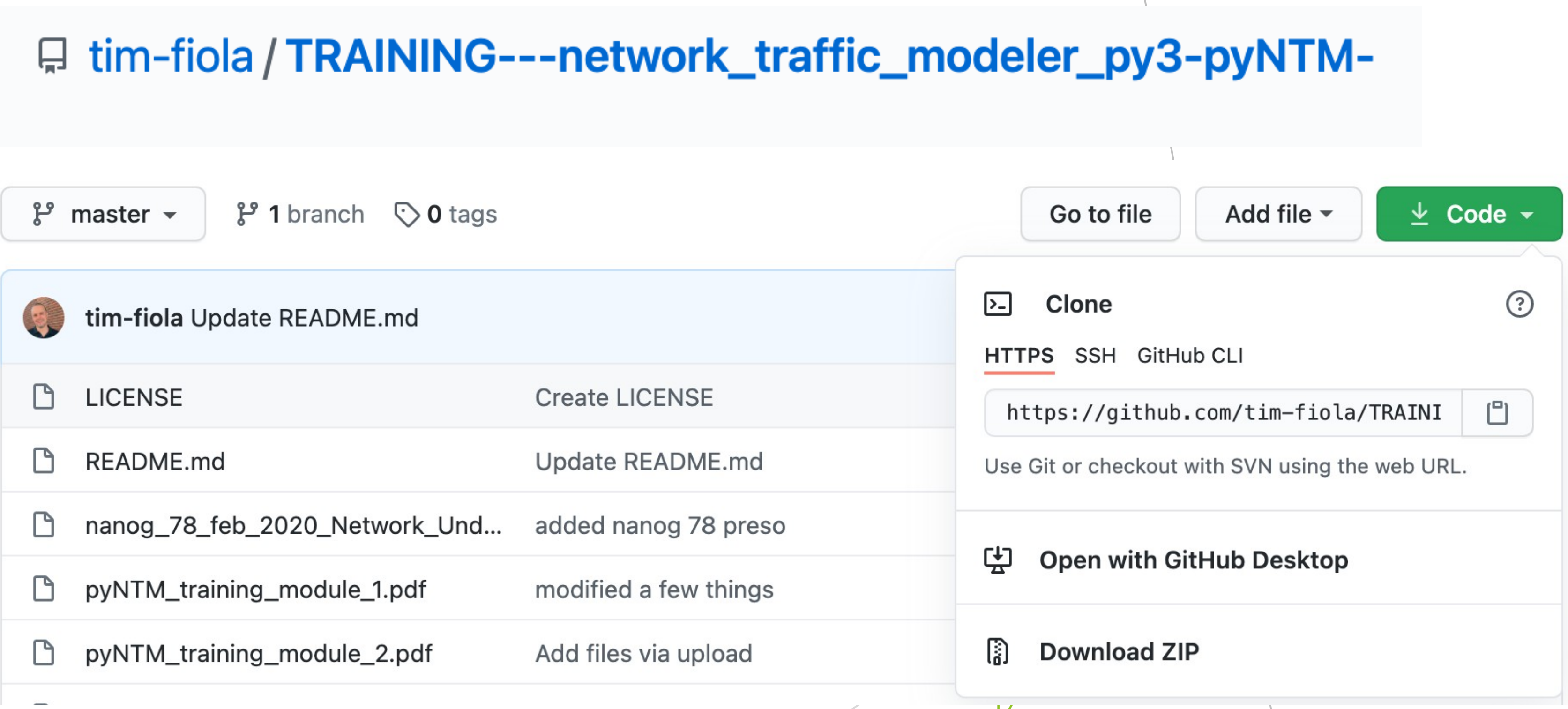
- There are 2 Model classes
- Both Classes support
  - *RSVP manual metrics*
  - *RSVP auto-bandwidth*
  - *Static (manually configured) RSVP LSP setup bandwidth*
  - *Setting a % of an Interface's bandwidth that can be reserved for RSVP LSPs (default is 100%)*
- PerformanceModel Class
  - *Supports only a single Circuit between any 2 Nodes*
  - *Does NOT support*
    - IGP shortcuts
    - Multiple Circuits between Nodes
  - *Will converge a bit faster than the FlexModel Class at the cost of supporting less topology features*
- FlexModel Class
  - *Supports IGP shortcuts*
  - *Supports multiple Circuits between Nodes*
  - *Requires the user to supply a Circuit ID value to match up the component Interfaces*
  - *May take longer to converge, but supports more topology features*

Exercise setup

J

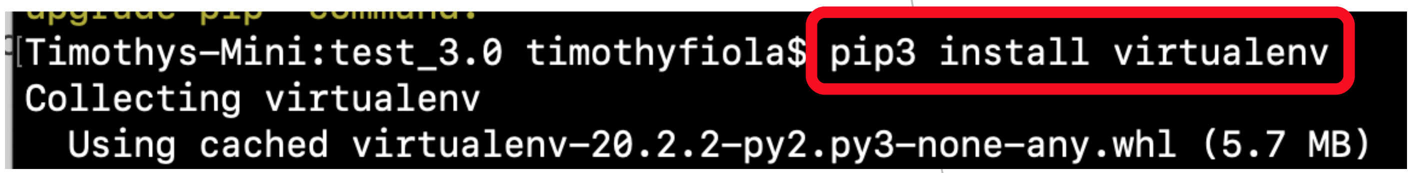
Copy the repository zip file to a practice directory and unzip it

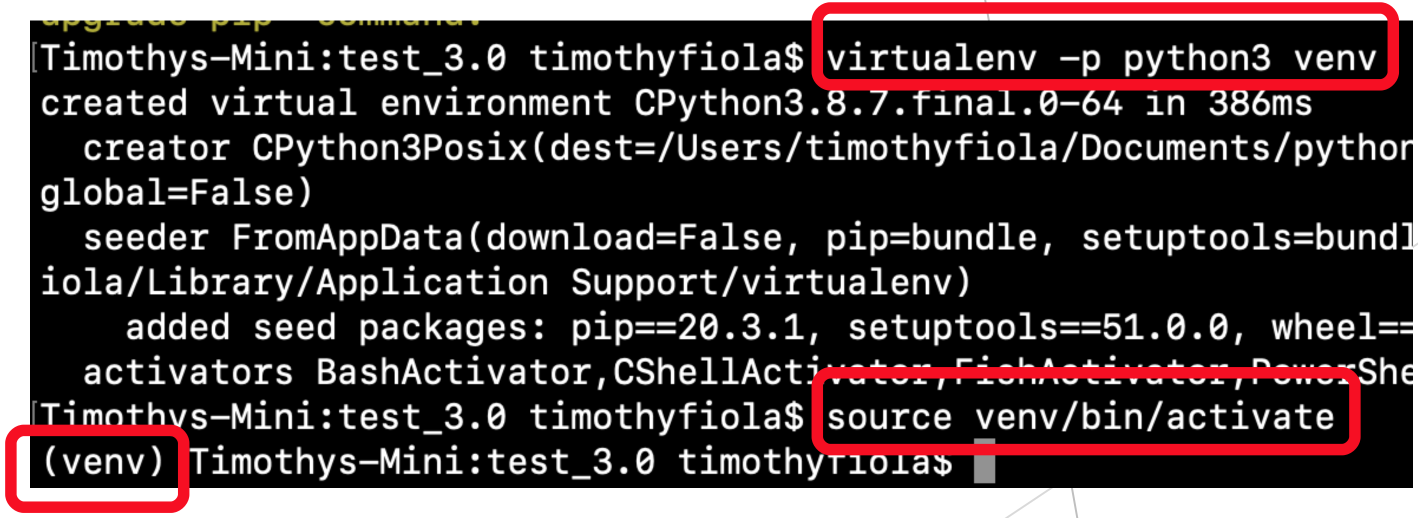
- The pyNTM Training Repository will have
  - The model data files used in the trainings
  - Examples of the scripts in the trainings
- As needed, move the specified model data files into your practice environment, which will be created in the next slide



Create virtual environment (recommended, optional)

- Navigate to a local directory
- Install the python *virtualenv* module
  - pip3 install virtualenv
- Set up the virtual environment
  - virtualenv -p python3 venv
- Start the virtual environment
  - source venv/bin/activate
- Notice the venv has been established





Let's get started!

- Install pyNTM (*pip3 install pyntm*)
- If you want to use the new WeatherMap visualization (beta), also explicitly install the *dash* and *dash-cytoscape* modules
- Start python3
- Import the PerformanceModel object
- Load Model from data file
  - *sample_network_model_file.csv* has Interfaces, Nodes, and Demands
  - IGP only/no RSVP LSPs in the file
- Observe node objects

```
pip3 install pyntm
```

```
pip3 install dash
```

```
pip3 install dash-cytoscape
```

```
(venv) Timothys-Mini:test_3.0 timothyfiola$ python3
Python 3.8.7 (v3.8.7:6503f05dd5, Dec 21 2020, 12:45:15)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> from pyNTM import PerformanceModel
>>>
>>> model1 = PerformanceModel.load_model_file('sample_network_model_file.csv')
RSVP_LSP_TABLE not in file; no LSPs added to model
>>>
>>> model1
PerformanceModel(Interfaces: 28, Nodes: 8, Demands: 11, RSVP_LSPs: 0)
>>>
>>> model1.node_objects
{Node('E'), Node('C'), Node('D'), Node('F'), Node('H'), Node('A'), Node('G'), Node('B')}
>>>
```

Converging the model

- In order to route the traffic across the network topology to get modeling data, the Model must be explicitly converged
- A Model should be converged after it is loaded from a file
- A Model should be re-converged if you make any change to the topology
  - Failing an Interface or Node
  - Un-failing an Interface or Node
  - New Node, Interface, traffic Demand, RSVP LSP, SRLG etc
  - Any change, otherwise, to the topology or traffic matrix
- Use the *update_simulation()* call on the Model object to converge the model
  - This will converge the model and run some internal validation checks

```
>>> model1.update_simulation()
Routing the LSPs . . .
LSPs routed (if present) in 0:00:00.001092; routing demands now . . .
Demands routed in 0:00:00.007133; validating model . . .
>>>
```

Visualization (beta) – a quick primer

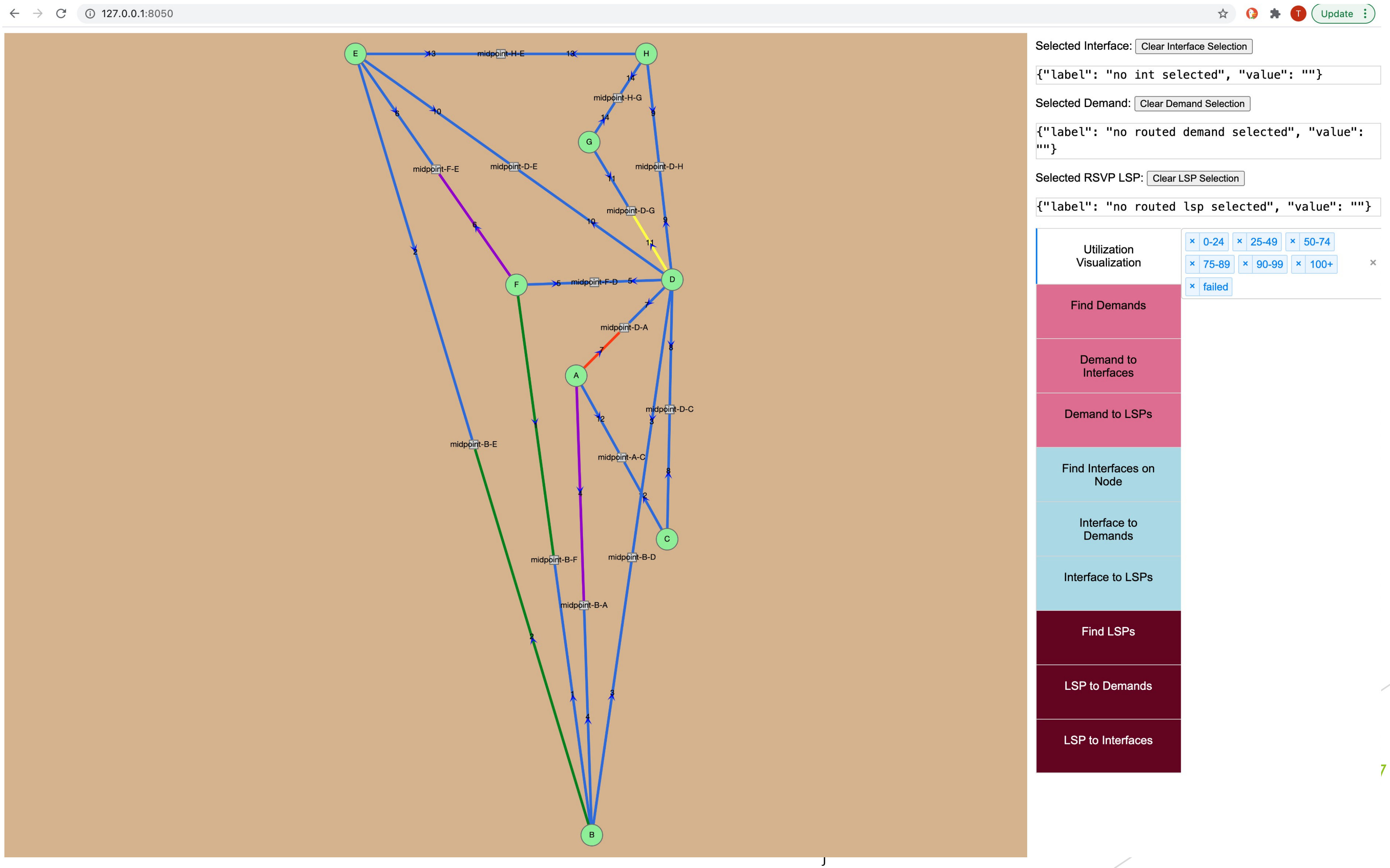
- Visualizing the topology now will help you understand what's going on in the subsequent exercises
- Import the WeatherMap via *pyNTM.weathermap*
- Make sure the model is converged! (if it's not, the visualization may not work)
- Create *WeatherMap* object
- Call the *create_weathermap* method
- Visualization found at http://127.0.0.1:8050/
- **Use CTRL+C in the terminal to exit the visualization**
- There is a thorough WeatherMap training module later in this presentation

```
>>> from pyNTM.weathermap import WeatherMap
>>>
>>> model1.update_simulation()
Routing the LSPs . . .
LSPs routed (if present) in 0:00:00.000693; routing demands now . . .
Demands routed in 0:00:00.003422; validating model . . .
>>>
>>> wm = WeatherMap(model1)
>>>
>>> wm.create_weathermap()

*** 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/
```

```
127.0.0.1 - - [11/Jan/2021 16:23:09] "POST /_dash-update-component HTTP/1.1" 200 -
^C<dash.dash.Dash object at 0x7fa5200361f0>
>>>
```

Visualization (beta) - continued

- Here's the visualization of *sample_network_model_file.csv*



This was just a quick primer on visualization, to allow you to see the topology you are working with.

There is a full training module on the WeatherMap visualization as well.

Feel free to click around with this one.

To reset the visualization, just refresh the page.

Live Exercise 1: Shortest path(s)

- Observe shortest path(s)
  - If there are multiple shortest paths, the 'path' value list will have multiple lists, each one a unique path
  - Each path list in the 'path' value list is an (ordered) list of Interfaces from source to destination
- In the example below, the shortest path from Node C to Node E
  - has a total IGP cost of 8
  - has a single shortest path
  - egresses 3 Interfaces, in order, from source to destination:
    - Node C to Node A
    - Node A to Node B

```
>>> from pprint import pprint
>>> sp_c_e = model1.get_shortest_path('C', 'E')
>>>
>>> pprint(sp_c_e)
{'cost': 8,
 'path': [[Interface(name = 'C-to-A', cost = 1, capacity = 200, node_object = Node('C'), remote_node_object = Node('A'), circuit_id = 3),
           Interface(name = 'A-to-B', cost = 4, capacity = 100, node_object = Node('A'), remote_node_object = Node('B'), circuit_id = 2),
           Interface(name = 'B-to-E', cost = 3, capacity = 200, node_object = Node('B'), remote_node_object = Node('E'), circuit_id = 11)]]}
```

A quick look at interface utilization

- Interface objects have several attributes and methods
- Quickly find the utilization of each interface with a simple *for* loop!
  - Interface.name
    - Name of the Interface
  - Interface.node_object.name
    - Name of the Node object that the Interface resides on
  - Interface.utilization
    - % traffic utilization
  - Easily add qualifiers to find Interfaces with specific attributes

```
>>> from pyNTM import Interface
>>> dir(Interface)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__format__', '__
ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_subclass__', '__le__'
, '__lt__', '__module__', '__ne__', '__new__', '__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__sizeof__', '__str__', '__subclasshook__', '__weakref__', '_key', 'add_to
_srlg', 'capacity', 'cost', 'demands', 'fail_interface', 'failed', 'get_circuit_object', '
get_remote_interface', 'lsps', 'remove_from_srlg', 'reservable_bandwidth', 'reserved_bandw
idth', 'srlgs', 'unfail_interface', 'utilization']
>>>
```

```
>>> for interface in model1.interface_objects:
...     print(interface.name, interface.node_object.name, interface.utilization)
...
A-to-D A 80.0
B-to-D B 7.5
C-to-A C 10.0
F-to-E F 105.0
H-to-D H 0.0
F-to-D F 0.0
B-to-F B 11.0
C-to-D C 0.0
B-to-E B 45.0
D-to-G D 60.0
A-to-B A 136.0
F-to-B F 25.0
B-to-A B 20.0
D-to-C D 0.0
E-to-F E 11.0
D-to-B D 0.0
A-to-C A 10.0
E-to-D E 15.0
G-to-D G 0.0
D-to-E D 0.0
E-to-B E 22.5
H-to-E H 0.0
D-to-F D 0.0
D-to-H D 0.0
H-to-G H 0.0
E-to-H E 20.0
G-to-H G 0.0
D-to-A D 6.67
>>>
```

```
>>> for interface in model1.interface_objects:
...     if interface.utilization >= 90:
...             print(interface.name, interface.node_object.name, interface.utilization)
...
F-to-E F 105.0
A-to-B A 136.0
>>>
```

Live Exercise 2: Failing an Interface/Circuit

- Circuit objects have two member Interface objects
  - One of the Interfaces fails, the Circuit and the remote Interface objects also fail
- Step 1: get Interface object
  - There are a few ways to do this
  - Technique 1: get interface via *get_interface_object* Model call
    - Need to know Node name and Interface name

```
>>> help(model1.get_interface_object)
```

```
Help on method get_interface_object in module pyNTM.master_model:

get_interface_object(interface_name, node_name) method of pyNTM.performance_model.PerformanceModel instance
    Returns an interface object for specified node name and interface name

    :param interface_name: name of Interface
    :param node_name: name of Node
    :return: Specified Interface object from self
(END)
```

```
>>> int_a_b = model1.get_interface_object('A-to-B', 'A')
>>> int_a_b
Interface(name = 'A-to-B', cost = 4, capacity = 100, node_object = Node('A'), remote_node_object = Node('B'), address = 12)
```

Step 1 (continued) – get Interface Object (technique 2)

- Get an interface object from a Node's interface list
  - Only need to know Node name
- Get the Node object
- List the Node's Interfaces
- Select the Interface from the list via its index
  - The index for a particular interface may change after an update_simulation call, so this is not the recommended programmatic way

```
>>> node_a = model1.get_node_object('A')
>>>
>>> node_a.interfaces(model1)
[Interface(name = 'A-to-D', cost = 8, capacity = 150, node_object = Node('A'), remote_no
de_object = Node('D'), address = 10), Interface(name = 'A-to-B', cost = 4, capacity = 10
0, node_object = Node('A'), remote_node_object = Node('B'), address = 2), Interface(name
 = 'A-to-C', cost = 1, capacity = 200, node_object = Node('A'), remote_node_object = Nod
e('C'), address = 11)]
>>>
>>> int_a_b_via_node_a = node_a.interfaces(model1)[0]
>>>
>>> int_a_b_via_node_a
Interface(name = 'A-to-D', cost = 8, capacity = 150, node_object = Node('A'), remote_nod
e_object = Node('D'), address = 10)
>>>
```

Live Exercise 2: Failing an Interface/Circuit (continued)

- Step 2: fail the interface object
  - Uses Interface *fail_interface* method

```
>>> int_a_b.fail_interface(model1)
>>>
```

- Step 3: update the simulation

```
>>> model1.update_simulation()
Routing the LSPs . . .
LSPs routed (if present) in 0:00:00.002986; routing demands now . . .
Demands routed in 0:00:00.012983; validating model . . .
>>>
```

- Step 4: check interface utilization

```
>>> for interface in model1.interface_objects:
...     print(interface.name, interface.node_object.name,
                interface.utilization)
...
A-to-D A 164.0
B-to-D B 7.5
C-to-A C 5.0
F-to-E F 105.0
H-to-D H 0.0
F-to-D F 0.0
B-to-F B 0.0
C-to-D C 6.67
B-to-E B 7.5
D-to-G D 60.0
A-to-B A Int is down
F-to-B F 25.0
B-to-A B Int is down
D-to-C D 6.67
E-to-F E 0.0
D-to-B D 12.5
A-to-C A 5.0
E-to-D E 35.0
G-to-D G 0.0
D-to-E D 69.0
E-to-B E 25.0
H-to-E H 0.0
D-to-F D 22.0
D-to-H D 20.0
H-to-G H 0.0
E-to-H E 0.0
G-to-H G 0.0
D-to-A D 13.33
>>>
```

Live Exercise 3: Finding traffic demands on an interface

- Looking at the interface utilizations from Exercise 2, the interface from Node A to Node D shows 164% utilized
- Let's see which Demands (traffic) is driving that utilization
- Use the Interface *demands* method

```
>>> int_a_d = model1.get_interface_object('A-to-D', 'A')
>>>
>>> dir(int_a_d)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '__form
at__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__', '__init_s
ubclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__', '__reduce__',
 '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '__str__', '__subclas
shook__', '__weakref__', '_capacity', '_cost', '_failed', '_key', '_srlgs', 'add
_to_srlg', 'address', 'capacity', 'cost', 'demands', 'fail_interface', 'failed',
 'get_circuit_object', 'get_remote_interface', 'in_ckt', 'lsps', 'name', 'node_o
bject', 'remote_node_object', 'remove_from_srlg', 'reservable_bandwidth', 'reser
ved_bandwidth', 'srlgs', 'traffic', 'unfail_interface', 'utilization']
>>>
>>> help(int_a_d.demands(model1))
```

```
>>> help(int_a_d.demands)
```

```
Help on method demands in module pyNTM.interface:

demands(model) method of pyNTM.interface.Interface instance
    Returns list of demands that egress the interface

    :param model: model object containing self
    :return: list of Demand objects egressing self
(END)
```

```
>>> for interface in model1.interface_objects:
...     print(interface.name,
...               interface.node_object.name,
...               interface.utilization)
...
B-to-F B 0.0
D-to-A D 13.33
D-to-F D 22.0
A-to-C A 5.0
D-to-C D 6.67
D-to-H D 20.0
D-to-B D 12.5
E-to-F E 0.0
H-to-E H 0.0
E-to-B E 25.0
G-to-D G 0.0
F-to-D F 0.0
G-to-H G 0.0
F-to-B F 25.0
E-to-D E 35.0
C-to-A C 5.0
A-to-D A 164.0
B-to-E B 7.5
H-to-D H 0.0
D-to-E D 69.0
F-to-E F 105.0
C-to-D C 6.67
B-to-D B 7.5
B-to-A B Int is down
D-to-G D 60.0
E-to-H E 0.0
A-to-B A Int is down
H-to-G H 0.0
>>>
```

Live Exercise 3: Finding traffic demands on an interface (continued)

- There are 6 demands on the Interface

```
>>> dmds_int_a_d = int_a_d.demands(model1)
>>>
>>> pprint(dmds_int_a_d)
[Demand(source = A, dest = H, traffic = 20, name = "''"),
 Demand(source = C, dest = E, traffic = 20, name = "''"),
 Demand(source = A, dest = B, traffic = 50, name = "''"),
 Demand(source = A, dest = F, traffic = 22, name = "''"),
 Demand(source = A, dest = E, traffic = 24, name = "''"),
 Demand(source = A, dest = D, traffic = 120, name = "''")]
>>>
```

Live Exercise 4: Finding Demand path(s) in an IGP network

- From our Demand results in Exercise 3, find the path of *Demand(source = A, dest = B, traffic = 50, name = "''")*
  - Keep in mind that the Circuit between Nodes A and B is failed (from Exercise 2)
- Get the demand object
  - Either by getting item at index 2 from *dmds_int_a_d* (list) (from Exercise 3)
    - The specific index for this demand will vary; *check your own results*

```
>>> dmd_a_b = dmds_int_a_d[2]
>>> dmd_a_b
Demand(source = A, dest = B, traffic = 50, name = "''")
>>>
```

  - Or using the Model *get_demand_object* method

```
Help on method get_demand_object in module pyNTM.model:

get_demand_object(source_node_name, dest_node_name, demand_name='none') method o
f pyNTM.model.Model instance
    Returns demand specified by the source_node_name, dest_node_name, name;
    throws exception if demand not found
(END)
```

```
>>> dmd_a_b = model1.get_demand_object('A', 'B', "''")
>>> dmd_a_b
Demand(source = A, dest = B, traffic = 50, name = "''")
>>>
```

Live Exercise 4: Finding Demand path(s) in an IGP network (continued)

- The demand has 2 ECMP paths
- The Demand path call returns a list of lists
  - Each component list shows the Interfaces the Demand egresses, in order, from the source Node to the Destination Node
- Keep in mind that we previously failed the Circuit between Nodes A and B

```
>>> int_a_b.failed
True
```

```
>>> dmd_a_b
Demand(source = A, dest = B, traffic = 50, name = "''")
>>>
>>> dir(dmd_a_b)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__', '__eq__', '_
_format__', '__ge__', '__getattribute__', '__gt__', '__hash__', '__init__',
 '__init_subclass__', '__le__', '__lt__', '__module__', '__ne__', '__new__'
, '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__', '
__str__', '__subclasshook__', '__weakref__', '_key', '_path_detail', 'dest_
node_object', 'name', 'path', 'path_detail', 'source_node_object', 'traffic
']
>>>
```

```
>>> len(dmd_a_b.path)
2
```

```
>>> for path in dmd_a_b.path:
...     pprint(path)
...     print()
...
[Interface(name = 'A-to-D', cost = 8, capacity = 150, node_object = Node('A'),
remote_node_object = Node('D'), circuit_id = 7),
 Interface(name = 'D-to-B', cost = 7, capacity = 200, node_object = Node('D'),
remote_node_object = Node('B'), circuit_id = 3)]

[Interface(name = 'A-to-D', cost = 8, capacity = 150, node_object = Node('A'),
remote_node_object = Node('D'), circuit_id = 7),
 Interface(name = 'D-to-E', cost = 4, capacity = 100, node_object = Node('D'),
remote_node_object = Node('E'), circuit_id = 10),
 Interface(name = 'E-to-B', cost = 3, capacity = 200, node_object = Node('E'),
remote_node_object = Node('B'), circuit_id = 2)]

>>>
```

Live Exercise 4: Finding Demand path(s) in an IGP network (continued) – *path_detail* attribute

- New in 3.0 and above, the *path_detail* attribute gives more detail on the demand path
- The *splits* entry provides a lot of insight!
  - A path object's *split* indicates how many times the demand object has split prior to transiting that object
  - Dividing a demand's *traffic* by a path object's *split* tells you how much of the demand traffic is on that object
    - Ex: The interface 'D-to-E' has 50/2 = 25 units of dmd_a_b's *traffic*
  - path_traffic tells how much of the demand's *traffic* is on that unique path
    - It is the demand's *traffic* divided by the path's highest *split* value

```
>>> dmd_a_b.traffic
50
```

```
>>> pprint(dmd_a_b.path_detail)
{'path_0': {'interfaces': [Interface(name = 'A-to-D', cost = 8, capacity = 150, node_object = Node('A'), remote_node_object = Node('D'), circuit_id = 1),
                           Interface(name = 'D-to-B', cost = 7, capacity = 200, node_object = Node('D'), remote_node_object = Node('B'), circuit_id = 6)],
            'path_traffic': 25.0,
            'splits': {Interface(name = 'A-to-D', cost = 8, capacity = 150, node_object = Node('A'), remote_node_object = Node('D'), circuit_id = 1): 1,
                       Interface(name = 'D-to-B', cost = 7, capacity = 200, node_object = Node('D'), remote_node_object = Node('B'), circuit_id = 6): 2}},
 'path_1': {'interfaces': [Interface(name = 'A-to-D', cost = 8, capacity = 150, node_object = Node('A'), remote_node_object = Node('D'), circuit_id = 1),
                           Interface(name = 'D-to-E', cost = 4, capacity = 100, node_object = Node('D'), remote_node_object = Node('E'), circuit_id = 7),
                           Interface(name = 'E-to-B', cost = 3, capacity = 200, node_object = Node('E'), remote_node_object = Node('B'), circuit_id = 11)],
            'path_traffic': 25.0,
            'splits': {Interface(name = 'A-to-D', cost = 8, capacity = 150, node_object = Node('A'), remote_node_object = Node('D'), circuit_id = 1): 1,
                       Interface(name = 'D-to-E', cost = 4, capacity = 100, node_object = Node('D'), remote_node_object = Node('E'), circuit_id = 7): 2,
                       Interface(name = 'E-to-B', cost = 3, capacity = 200, node_object = Node('E'), remote_node_object = Node('B'), circuit_id = 11): 2}}}
```

Live Exercise 5: Unfailing an Interface

- Now that the failure analysis is complete, unfail the Interface on Node A facing Node B
  - The Interface on Node B facing Node A also unfails automatically

```
>>> int_b_a = model1.get_interface_object('B-to-A', 'B')
>>>
>>> int_b_a.failed
True
>>>
>>> int_a_b.failed
True
```

```
>>> int_a_b.unfail_interface(model1)
>>>
>>> int_a_b.failed
False
>>>
>>> int_b_a.failed
False
>>>
>>> model1.update_simulation()
Routing the LSPs . . .
LSPs routed (if present) in 0:00:00.001467; routing demands now . . .
Demands routed in 0:00:00.005815; validating model . . .
>>>
```

FIN

J