Open and virtualized networks - Network Exercises - Part 10

Network Exercise

The aim of these exercises is to introduce in the Network abstraction the concept of traffic matrix and the request of a specific bitrate (traffic grooming size) for a connection request.

1. Modify the Network class such that the number of channels available in each line is given once as an attribute `Nch` and set accordingly throughout the software. Set `Nch` to 20.

   If too slow in the following exercises, you can try set Nch to 10. In main():

   ```
   ..
   network = Network('../nodes.json', nch=10)
   ..
   ```

   In Network class:

   ```
   class Network(object):
       def __init__(self, json_path, nch=10):
           node_json = json.load(open(json_path, 'r'))
           self._nodes = {}
           self._lines = {}
           self._connected = False
           self._weighted_paths = None
           self._route_space = None
           self._Nch = nch
           for node_label in node_json:
               # Create the node instance
               node_dict = node_json[node_label]
               node_dict['label'] = node_label
               node = Node(node_dict)
               self._nodes[node_label] = node

               # Create the line instances
               for connected_node_label in node_dict['connected_nodes']:
                   line_dict = {}
                   line_label = node_label + connected_node_label
                   line_dict['label'] = line_label
                   node_position = np.array(node_json[node_label]['position'])
                   connected_node_position = np.array\
                       (node_json[connected_node_label]['position'])
                   line_dict['length'] = np.sqrt\
                           (np.sum((node_position - connected_node_position) ** 2))
                   line_dict['Nch'] = self.Nch

                   line = Line(line_dict)
   ```

```
                self._lines[line_label] = line


    @property
    def Nch(self):
        return self._Nch


    def set_weighted_paths(self):

            ....

        for i in range(self.Nch):
            route_space[str(i)] = ['free'] * len(paths)
        self._route_space = route_space

        ...
```

In Line class:

```
class Line(object):
    def __init__(self, line_dict):
        ...
        self._Nch = line_dict['Nch']
        self._state = ['free'] * self._Nch
        ...

    def nli_generation(self, signal_power, rs, df, Bn=12.5e9):
        Pch = signal_power
        loss = np.exp(- self.alpha * self.span_length)
        N_spans = self.amplifiers
        eta = 16 / (27 * pi) * \
              self.gamma ** 2 / (4 * self.alpha * self.beta * rs ** 3) * \
              np.log(pi ** 2 * self.beta * rs ** 2 * self.Nch **\
                    (2 * rs / df) / (2 * self.alpha))

              ...
```

2. In your main program, perform a Monte-Carlo simulation on the traffic matrix with N_{MC} realizations. Keep the traffic matrix to uniform with constant request_rate for each node pair. Randomize instead the order in which the connection requests between each couple of nodes of the traffic matrix are issued. (You can use the shuffle() function to randomize the node pairs). For each Monte-Carlo realization start with the empty network and load it with a different node pair sequence of connection requests. Start with $N_{MC} = 100$. Assume `shannon` as transceiver technology.

```
def main():
    NMC = 5
    node_pairs_realizations = []
    stream_conn_list = []
    lines_state_list = []

    for i in range(NMC):
        print('Monte-Carlo Realization #{:d}'.format(i+1))

        network = Network('../nodes.json', nch=10, upgrade_line='AC')
# creates nodes and line objects
        network.connect()  # connects the net by setting the line \
                successive attribute with the node object at the end of the line
        network.draw()

        node_labels = list(network.nodes.keys())
        T = create_traffic_matrix(node_labels, 400, multiplier=5)
        t = T.values
        # print(T)

        node_pairs = list(filter(lambda x: x[0] != x[1], list(it.product(node_lal

        shuffle(node_pairs)  # Create allocation request sequence realization
        node_pairs_realizations.append(copy.deepcopy(node_pairs))
        connections = []
        for node_pair in node_pairs:
            connection = Connection(node_pair[0], node_pair[-1], \
                    float(T.loc[node_pair[0], node_pair[-1]]))
# creates connection object between random node pair
            connections.append(connection)  # list of connection objects

        streamed_connections = network.stream(connections, best='snr', \
                transceiver='shannon')

        stream_conn_list.append(streamed_connections)
        lines_state_list.append(network.lines)  # Get lines state

        ...
```

3. Calculate the average traffic in the network and the average bitrate per lightpath. Obtain these results as the average of the traffic and the average bitrate obtained in each different Monte-Carlo realization.

```
def main():

    ....

    # Get MC stats
    snr_conns = []
    rbl_conns = []
    rbc_conns = []

    for streamed_conn in stream_conn_list:
        snrs = []
        rbl = []

        [snrs.extend(connection.snr) for connection in streamed_conn]

        for connection in streamed_conn:
            for lightpath in connection.lightpaths:
                rbl.append(lightpath.bitrate)

        rbc = [connection.calculate_capacity() for connection in streamed_conn]

        snr_conns.append(snrs)
        rbl_conns.append(rbl)
        rbc_conns.append(rbc)


    avg_snr_list = []
    avg_rbl_list = []
    traffic_list = []

    [traffic_list.append(np.sum(rbl_list)) for rbl_list in rbl_conns]
    [avg_rbl_list.append(np.mean(rbl_list)) for rbl_list in rbl_conns]
    [avg_snr_list.append(np.mean(list(filter(lambda x: x != 0, snr_list))))\
                for snr_list in snr_conns]

    print('\n')
    print('Line to upgrade: {}'.format(max(avg_congestion, \
                    key=avg_congestion.get)))
    print('Avg Total Traffic: {:.2f} Tbps'.format(np.mean(traffic_list) * 1e-3))
    print('Avg Lighpath Bitrate: {:.2f} Gbps'.format(np.mean(avg_rbl_list)))
    print('Avg Lighpath SNR: {:.2f} dB'.format(np.mean(avg_snr_list)))
```

4. Calculate also the average congestion of the network lines. The congestion of a line is defined as the percentage of allocated channel to the total available channel `Nch`. Calculate the congestion for each line of the Network and average the result over all the Monte-Carlo iterations. Plot the average congestion at the end of the Monte-Carlo simulation as an histogram with the lines on the X-axis and its average congestion on the Y-axis

```
def main():

    ....

    # Congestion
    lines_labels = list(lines_state_list[0].keys())
    congestions = {label: [] for label in lines_labels}

    for line_state in lines_state_list:
        for line_label, line in line_state.items():
            cong = line.state.count('occupied') / len(line.state)
```

```
                congestions[line_label].append(cong)

        avg_congestion = {label: [] for label in lines_labels}
        for line_label, cong in congestions.items():
            avg_congestion[line_label] = np.mean(cong)

        plt.bar(range(len(avg_congestion)), \
                        list(avg_congestion.values()), align='center')
        plt.xticks(range(len(avg_congestion)), list(avg_congestion.keys()))
        plt.show()

        ...
```

5. set a rate_request multiplier on the traffic matrix. It is an integer number multiplying the values of the traffic matrix in order to increase the number of lightpath that needs to be allocated for each connection request between a node pair. Tune this parameter so that the lightpaths requests are enough to saturate the network. Tune also the N_{MC}, by increasing its value to 200, 300, 500 and so on until the average rate per lightpath converges to an almost constant value. Look what happens to the congestion as you increase the rate multiplier.

```
def create_traffic_matrix(nodes, rate, multiplier=1):
    s = pd.Series(data=[0.0] * len(nodes), index=nodes)
    df = pd.DataFrame(float(rate*multiplier), \
                        index=s.index, columns=s.index, dtype=s.dtype)
    np.fill_diagonal(df.values, s)

    return df
```

6. Look at the congestion plots and identify the most congested line. Write a method in the **Network** class called **upgrade_line()** which takes a line object (or its string description) and perform an upgrade of the equipment by decreasing by 3 dB the noise figure of the amplifiers only for that line. Redo the Monte-Carlo simulation on the upgraded network and check for improvements in congestion and overal network capacity.

```
class Network(object):
    def __init__(self, json_path, nch=10, upgrade_line=''):
        node_json = json.load(open(json_path, 'r'))

        ....

        # if specified, upgrade line
        if not upgrade_line == '':
            self.lines[upgrade_line].noise_figure \
                = self.lines[upgrade_line].noise_figure - 3
```