1_NetworkSetup

April 20, 2022

1 Network Analysis Using Reinforcement Learning

1.1 Spring 2022: Shaoyu Pei, Avery Peiffer

1.2 Advisor: Dr. Mai Abdelhakim

1.3 Notebook 1: Introduction and Network Setup

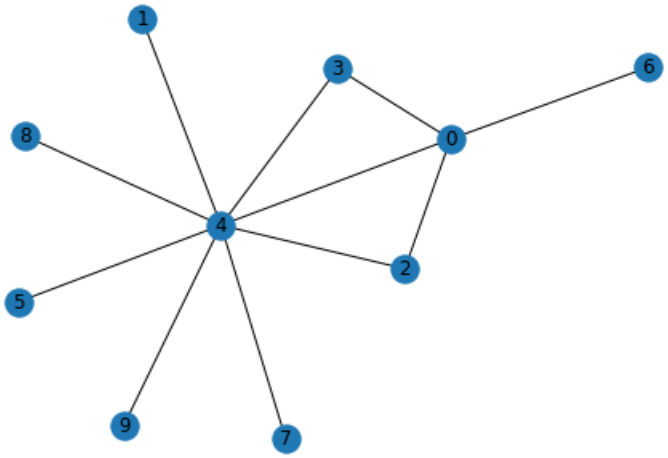
The purpose of this notebook is to show you how we define an IoT network using the `networkx` package. For this study, we are interested in the idea that devices can manipulate data transmitted in packets as it is being forwarded from a source device to a destination device. This would result in incorrect data arriving at the destination device. This may happen intentionally, as a result of a cyber attack on the device, or unintentionally. We refer to these types of devices as malicious, because they can compromise the integrity of the packets that are sent throughout the network. Overall, we want to use reinforcement learning to create an algorithm that can identify which devices are likely to be malicious, so that we can avoid them when transmitting data through the network.

Please refer to Dr. Abdelhakim's paper in `../papers/` for more details, as that is the paper that forms the basis of this work.

Note: If you get any errors due to packages not being installed, please add them to the cell below. We might have not encountered these errors due to already having those packages installed on our computers for other courses/projects/etc. You can install a package with: `!pip3 install <package-name>`

[1]:
```
import random
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
```

We represent a network as a graph, made up of nodes and edges. A node represents a device and an edge represents a connection between two devices. We use a partial mesh topology to define the network, which means that not all pairs of nodes are directly connected to each other (which would be a full mesh topology). The following image shows an example of a network created using the `networkx` package.



Note: To keep things clear as we write code, we have a small convention in defining the graph. The source node is always Node 0, and the destination node is always the highest-numbered node (9, in this image).

`Networkx` allows us to create a random graph according to a set of parameters. We define those parameters here: * `network_size` - the number of devices in the network. * `edge_prob` - how likely it is that any two devices in the network are connected. `Edge_prob = 1` means that every device is connected to every other device; this isn't useful for this study, as we want to learn about the paths from one device to another. However, we want the graph to be connected; that is, there does exist at least one path from one device to another. For this reason, we generally start with a low `edge_prob`, then make use of a method to add edges to the network until it is connected. This will result in a sparse, but connected, graph.

We also use some parameters to determine which devices in the network are malicious, as well as their potency. * `percent_mal` - the percentage of malicious devices in the network. * `attack_probs` - how likely it is that the malicious devices will attack on a single transmission. We represent this as a list of two values, which form the uniform distribution from which the attack probability is drawn.

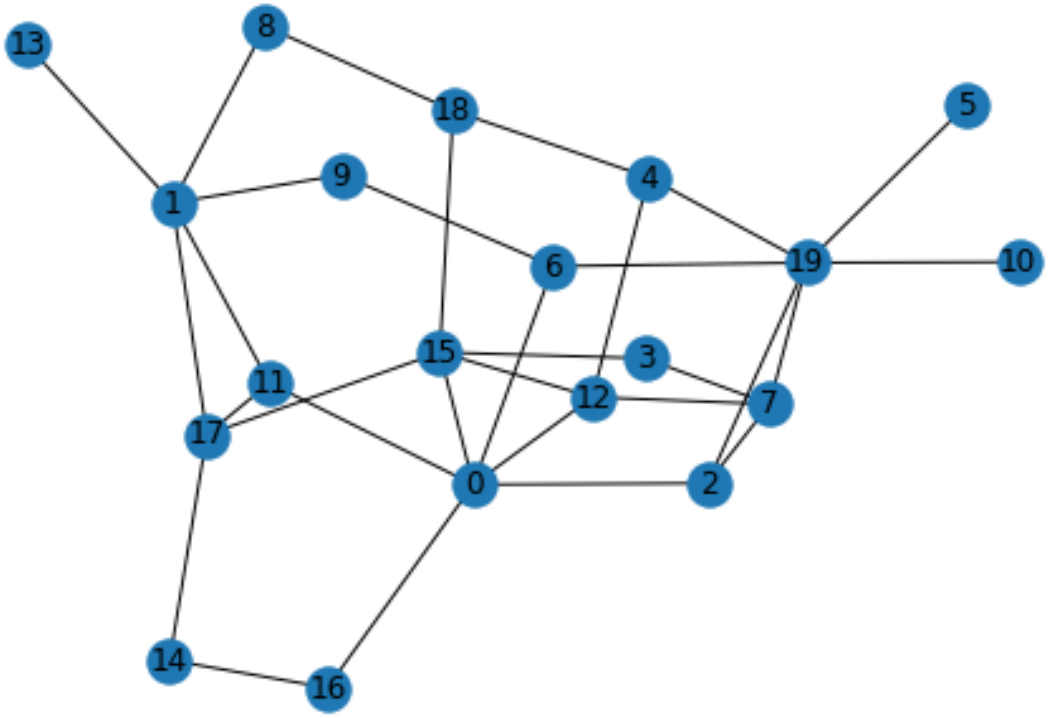
[2]:
```
network_size = 20
edge_prob = 0.1
percent_mal = 0.3
attack_probs = [0.2, 0.8]
```

Here, we create the graph according to the hyperparameters defined above. We start with a random graph; then, we add edges using the `random_edge` function until the graph is connected. Again, the source node is always node 0 and the destination node is always the highest-numbered node.

[3]:
```
def render(graph):
    nx.draw(graph, with_labels=True)
    plt.show()

def random_edge(graph):
    edges = list(graph.edges)
```

```
    nonedges = list(nx.non_edges(graph))

    chosen_edge = random.choice(edges)
    chosen_nonedge = random.choice([x for x in nonedges if chosen_edge[0] ==␣
↪x[0] or chosen_edge[0] == x[1]])

    graph.add_edge(chosen_nonedge[0], chosen_nonedge[1])

    return graph
```

[4]:
```
src = 0
dst = network_size - 1

G = nx.gnp_random_graph(network_size, edge_prob)

while not nx.is_connected(G):
    G = random_edge(G)

render(G)
```



At this point, we create a class to encode information for each node about whether it is malicious and, if so, what its attack probability is. We call this class `device`. We use a somewhat lazy class definition to accomplish this, as we can add attributes to the class later.

We then randomly assign some number (according to `percent_mal`) of the devices to be malicious. We will assume that the source and destination nodes can't be malicious, since that defeats the purpose of the study.

[5]:
```
class device:
    pass
```

[6]:
```
mal_nodes = []

num_mal = int(network_size * percent_mal)

while (num_mal > 0):
    rand = np.random.randint(0, network_size)

    if rand != src and rand != dst and rand not in mal_nodes:
        mal_nodes.append(rand)
        num_mal -= 1
```

For each node in the graph, we create a corresponding `device` class for it. Each device has the following attributes: * `node` - the label of the node in the graph. * `mal` - a Boolean flag indicating whether the device is malicious. * `attack_prob` - a random number drawn from the uniform distribution bounded by `attack_probs` above. This indicates how likely the device is to attack on any given transmission. If the device is not malicious, `attack_prob = 0`.

[7]:
```
nodes = list(range(0, network_size))
devices = []

for node in nodes:
    d = device()
    d.node = node

    if node in mal_nodes:
        d.mal = True
        d.attack_prob = np.random.uniform(attack_probs[0], attack_probs[1])
    else:
        d.mal = False
        d.attack_prob = 0

    devices.append(d)
```

To verify that our network is set up correctly, we can run a brief test. First, we will print out the malicious nodes in the graph.

[8]:
```
print('Malicious Nodes:')
for d in devices:
    if d.mal:
        print(f'{d.node} - attack probability is {d.attack_prob:.2f}.')
```

```
Malicious Nodes:
6 - attack probability is 0.68.
9 - attack probability is 0.39.
10 - attack probability is 0.32.
13 - attack probability is 0.33.
15 - attack probability is 0.68.
18 - attack probability is 0.73.
```

The graph's `all_simple_paths` attribute gives us all of the paths from the source node to the destination node (that only visit a node once). We will go through each path and determine if a device has corrupted the path. For each malicious node along the path, we generate a random number and compare it to the node's attack probability to see if it will attack the path.

The `is_corrupted` function represents how our network is actually able to tell whether or not a transmission was corrupted. In real life, this would be accomplished by the end device doing a hash operation on the incoming data and checking the result. In the future, this function could be changed to more closely resemble the real-life version, but still provides useful information in its current state.

Note: This test might take some time to run if the number of nodes is high.

[9]:
```
def is_corrupted(path, verbose=True):
    for node in path:
        attack_prob = devices[node].attack_prob
        attacked = random.uniform(0,1) < attack_prob

        if attacked:
#             if verbose:
#                 print('Path = ', path)
#                 print('Node %d attacked this path' % node)
            return True
    return False
```

[10]:
```
paths = list(nx.all_simple_paths(G, src, dst))
paths_corruption = []

for path in paths:
    paths_corruption.append(is_corrupted(path))
```

We also have support for loading a custom graph from the `./env` folder. The formatting of the environment files is relatively simple and is outlined in `./env/env_format.txt`. If you choose to load in predefined graphs, please make sure that the files are read in correctly and there are no typos anywhere. Try loading it here first to make sure that everything makes sense.

[11]:
```
def read_network(fname):
    with open(fname) as f:
        lines = [line.rstrip() for line in f]
        print(f'Lines: {lines}')
```

```
        network_size = int(lines[0])
        print(f'Network size: {network_size}')

        edges = [eval(x) for x in lines[1:-1]]
        print(f'Edges: {edges}')

        attack_probs = [float(x) for x in lines[-1].split(',')]
        print(f'Attack probabilities: {attack_probs}')
```

[12]:
```
read_network('./env/env1.txt')
```

```
Lines: ['10', '0,2', '2,1', '3,0', '3,1', '5,0', '5,1', '1,9', '0,4', '0,6',
'0,7', '0,8', '0,0,0,.64,0,.85,0,0,0,0']
Network size: 10
Edges: [(0, 2), (2, 1), (3, 0), (3, 1), (5, 0), (5, 1), (1, 9), (0, 4), (0, 6),
(0, 7), (0, 8)]
Attack probabilities: [0.0, 0.0, 0.0, 0.64, 0.0, 0.85, 0.0, 0.0, 0.0, 0.0]
```