Assignment 1

Eduardo López

We are going to use `python`, `networkx`, and `matplotlib` to tackle this and many of our assignments through this course. This brief introduction will serve as setting for the assignment to follow. In terms of formatting, computational instructions are going to be given as if working on the command line of a bare bones `python` interface. This can easily be ported into a interfaces such as jupyter notebooks.

1 Basic `networkx` instructions

Begin by starting a `python` session in your environment of choice. Once inside that session, to open `networkx` and perform some basic operations, proceed as follows

```
>>> import networkx as nx #Import networkx
>>> G=nx.Graph() # Create an undirected network G
>>> G.add_node(1) # Add node 1 to G
>>> G.add_node(2) # Add node 2 to G
>>> G.add_edge(1,2) # Add link between 1 and 2 to G
>>> G.add_edge(1,4) # Add link 1 to 4 (and create 4)
    to G
>>> G.add_edge(2,4) # Add link 2 to 4 to G
>>> G.add_edge(2,3) # Add link 2 to 3 (and create 3)
    to G
>>> G.add_edge(3,4) # Add link 3 to 4 to G
```

This listing creates the network in Fig. 1. `networkx` is capable of either creating nodes or creating links. When creating links, if a node does not exist, it implicitly creates it; if the node does exist, then it performs the necessary changes to the characteristics of the node. In the example, both approaches are illustrated.

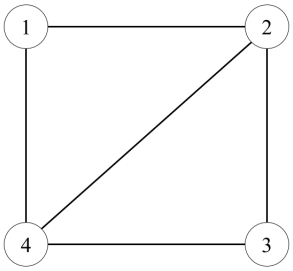

Figure 1: Network to illustrate basic functionality of `networkx`.

Let us learn about some of the functionalities of `networkx` before we present the assignment. The object `G` is a type of `networkx` object (a class) that comes with useful methods. For instance, we can determine information about the nodes in an easy way, such as their degrees or their neighbors

```
>>> G.degree(1) # Use method degree on G and node 1.
2
>>> G.degree(2) # Use method degree on G and node 2.
3
>>> G.degree(3) # Use method degree on G and node 3.
2
>>> G.degree(4) # Use method degree on G and node 4.
3
>>> list(G.neighbors(1)) # Method to find neighbors
   of 1
[2,4]
>>> list(G.neighbors(2)) # Method to find neighbors
   of 2
[1,3,4]
>>> list(G.neighbors(3)) # Method to find neighbors
   of 3
[2,4]
>>> list(G.neighbors(4)) # Method to find neighbors
   of 4
[1,2,3]
>>> G.has_edge(1,2) # Check if 1,2 are connected
```

```
True
>>> G.has_edge(2,1) # G is undirected so 1,2=2,1
True
>>> G.has_edge(1,3) # Check if 1,3 are connected
False
```

These simple commands show us some of the many useful properties that networks created with `networkx` posses.

One of the goals of this course is to understand the functions embedded into `networkx`. For this, we may need a small primer on creating a function in `python`. Note the indentation. It is very important. By the way, a function can return all kinds of thinks; for example, a network can be returned from a function. Let us make two simple functions, one of them relying on `networkx`

```
>>> # A function to use Pythagoran Theorem
>>> def HypotLen(x,y):
...     H2=x**2+y**2
...     H=H2**0.5
...     return(H)
>>> HypotLen(3,4) # Call HypotLen with inputs needed
5.0
>>> # A function to count links of dummy network H
>>> def m(H):
...     s=0
...     for i in H.nodes(): # H.nodes() = nodes of H
...         s=s+H.degree(i)
...     s=s/2 # Handshaking Lemma
...     return(s)
>>> m(G) # Apply m function to G define before
5
>>> G.size() # A direct networkx method to obtain m
5
>>> G.order() # A direct networkx method to obtain n
4
>>> list(G.nodes()) # python list of nodes of G
[1,2,3,4]
>>> list(G.edges()) # python list of links of G
[(1,2),(1,4),(2,3),(3,4),(2,4)]
>>> VG=list(G.nodes()) # save node list in VG
>>> EG=list(G.edges()) # save the link list in EG
```

```
>>> VG[0] # First (not 0th) element of VG
1
>>> VG[3] # Fourth (not 3rd) element of VG
4
>>> VG.index(3) # Find the index storing node 3
2
>>> VG[2] # Check
3
>>> EG[-1] # Last element of EG
(2,4)
```

Note that `networkx` has many many functions we can directly use. For a good reference of all that `networkx` can do, go to the `https://networkx.github.io/`. Note, for instance, methods such as `size(), order(), nodes(), edges()` all of which are applied directly onto a network `G`. By the way, notice how one can save lists of nodes, links, or anything else by creating some `python` list variables. Pay close attention to the indexing!

A python routine is called with the prefix, if any, that we may have chosen (in our case, we use `nx`). We saw this when defining `G` above. Also, for example

```
>>> A=nx.adjacency_matrix(G) # Adjacency matrix of G
>>> A[0,1] # are nodes 1 and 2 linked?
1
>>> A[0,2] # are nodes 1 and 3 linked?
0
```

Note the misalignment between locations in the matrix, which start at 0 and go up to $n-1$, with n the nodes of the network, versus the names of our nodes going from 1 ro n. These matrices *always* behave this way! Watch out.

We now have enough to proceed.

2 Assignment

This first assignment is *introductory* and it aims to put us all on the same page with regards to using `networkx` and `python`.

Instructions

1. In each problem or assignment, carefully pay attention to what you are allowed to use directly from `networkx` and what you will need to create from scratch.

2. When presenting the assignment, please have your laptop ready with all the necessary libraries, data, and code loaded, and the interface for `python` ready so that we can see on screen the code for the functions.

3. In general, the code that you create must be such that it can receive *any* new network `G` as input and produce the output needed. If, for instance, the code is supposed to produce the number of links of a network, then it should be able to receive any network (of the right type) as input, and provide as output the number of links.

Tasks

Important: after you write the first line of the functions, starting with `def ...`, on the next line add the instruction `import networkx as nx`. This will import `networkx` internally to the function so that if you forget to import `networkx` in your session, you will not get an error when running the function. On the other hand, your session *should also know how to deal with a network*, so you will have to also import `networkx` outside the function.

Now, the work

1. Create a function that takes as input a `networkx` object of type `Graph()` and two of its nodes `i,j` and outputs a 1 or a 0 if they are, respectively, connected or disconnected. In other words, create your own link indicator. Check it by applying it to `G` as defined above.

2. Using the link indicator in Prob. 1 and **not** the `degree` method for `Graph()` objects, create a function that takes as input a `networkx` object of type `Graph()` and one of its nodes `i`, and returns the degree of node `i`. Test it on `G` from above.

3. Create two functions that take as input a `networkx` object of type `Graph()` and generate a printout of each node and its degree in two ways (and thus the 2 functions):

   (a) by using `degree` from `networkx`, and

   (b) by using the function in Prob. 2.

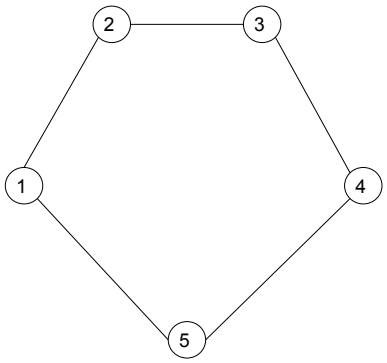

Figure 2: A ring network with $n = 5$.

The format of the printout can be very simple like two columns, with one column the node name and the other the corresponding degree. A statement like `print(i,ki)` where `ki` is the degree of `i` would be fine. Test this in `G` above.

4. Create a function that generates a network in the shape of a ring. An example can be found in Fig. 2. The network should be of type `Graph()`. The function should accept any size n as input. Check this function in a few ways:

   (a) by determining the relation between the number of nodes n and the number of links m of the resulting networks (check for instance, networks with $n = 100, 200, \ldots, 1000$), and

   (b) by determining the degree of various nodes of the result networks.

5. Create two functions that takes as input a `networkx` object of type `Graph()` and one of its nodes `i` and determine the number of v-shapes that visit `i`. Do this in two ways (hence two functions):

   (a) by using the `degree` method for `Graph()` of `networkx`, and

   (b) by using the function in Prob. 2 above and **not** the method `degree`.

   Test it on `G` from above, and on a ring network of size $n \geq 100$.

6. Create two functions that takes as input a `networkx` object of type `Graph()` and one of its nodes `i` and determine the number of triangles that visit `i`. Do this in two ways:

   (a) by using the `has_edge` method for `Graph()` of `networkx` and **not** by the `networkx` routine `triangles` (hint: it may help to use a list of neighbors of `i` and use some `python` list facilities, and match that with a `python` function that enumerates things or the `range` function), and

   (b) by using the `networkx` function `adjacency_matrix` (hint: remember that the numbering of the elements in the matrix and the names of nodes are not the same thing... maybe use list method like `index`?).

   Test it on `G` from above, and on a ring network of size $n \geq 100$.

7. Create a function that takes as input a `networkx` object of type `Graph()` and one of its nodes `i` and determines the local clustering of `i`. Construct this function by using the functions created in Probs. 5 and 6. Test it on `G` from above, and some nodes of a ring network of size $n \geq 100$. You can then check your result against the `networkx` function `clustering`.

8. Create a function that takes as input a `networkx` object of type `Graph()` and determines the global clustering of the network. Construct this function by using the functions created in Probs. 5 and 6. Test it on `G` from above, and on a ring network of size $n \geq 100$. You can then check your result against the `networkx` function `transitivity`.