Describing Code for the Subsequent Projects

Description

The purpose of this document is to explain all of the code found throughout the repository that isn't covered by the docstrings. Namely, much of the `.py` functions lack explanations as they exist to streamline code throughout the different notebooks. As such, each subsequent section here will represent a different `.py` file, including its unit testing.

generate_networks.py

The only item here, the class, is here to store all of the network properties and to include its basic methods, mostly that of rewiring. Of course, the imports come first:

```
from numpy import append, argwhere, arange, array, diag, delete, ones, random, sum, triu
from matplotlib import pyplot as plt
import networkx as nx
```

The purpose of importing `numpy` in portions like these is to reduce the time it takes to begin the notebooks. Namely, importing the entirety of the library takes time and space, so reducing it down to the only essential functions works wonderfully. `numpy` is used primarily for its vector operations and data structures.

Further, `matplotlib` has the same treatment, by only importing a subsection in `matplotlib.pyplot`. There are many plotting functions, so trying to reduce this down to only a few would pose its own challenge. The same can be said for `networkx`, so importing them in bulk takes time, but is a necessary sacrifice. Of course, both of these are used for their plotting capabilities.

Now, examining the actual class:

```
class Erdos_Renyi_GNP:
```

For our purposes, the graphs that we are using can be represented by an Erdos-Renyi graph, through some type of manipulation. As such, this class will act as the super class to any extended classes. So, it takes in no other classes since it is the parent.

init

Inside of this, once all of the initializations have been executed, there exists this portion

```
if A == None:
            self.A = triu(array(random.rand(N, N) < p, dtype = int))
            if self_edges == False:

                self.A = self.A + self.A.T - 2*diag(diag(self.A))

            else:
                self.A = self.A + self.A.T - diag(diag(self.A))

else:  self.A = A
```

which sets the adjacency matrix. Since the class allows for an adjacency matrix to be passed beforehand, namely useful if copying the methods, we check to see if this flag has been changed from its default `None` to some value. If it is, we assign the set value to the adjacency matrix. If not, we form our own.

Effectively, `random.rand(N, N)` creates a matrix of size $N \times N$, with values $A_{ij} \in [0, 1]$. If $A_{ij} < p$, where p is the edge probability, then that entry is replaced with a 1. Otherwise, it's set to 0. This is the result of `array(random.rand(N, N) < p, dtype = int)`. We take the `triu`, or upper triangular, of this result so that we can recombine it on the basis of self edges. Of course, when looking at something like the number of edges this introduces a form of error, but by doing this it forces symmetry.

Now, `self_edges` is a flag determining whether or not a node can be connected to itself. In our case, it never will. So, we add `self.A` with its transpose `self.A.T` and substract `2*diag(diag(self.A))`. Note that `diag(self.A)` grabs the diagonal elements of `self.A` and `diag(diag(self.A))` places those in the diagonal of a matrix of zeros with size $N \times N$. Since `triu` include the 1's along the diagonal, they're added twice and as such we subtract twice the diagonal.

Otherwise, we only subtract once due to the symmetry allowing for the diagonal to be a multiple of two, at least for an undirected and unweighted graph. To get the actual diagonal matrix, we execute

```
self.D = diag(sum(self.A, axis = 1))
self.L = self.D - self.A
```

Here, `sum(self.A, axis = 1))` sums the value of `self.A` across the second axis (since everything starts at 0). This could be done on the first axis as well, but for convention we use the second. This produces a vector of size N, which represents the diagonal values of `self.A`. We then use `diag` to force this vector into the diagonal of a zero matrix, like previously. Subtracting `self.D` and `self.A` is just the Laplacian `self.L`, as defined in the Documentation.

Last of the `init` function is declaring the edges and their locations:

```
self.M = sum(self.A)/2

self.edges = argwhere(triu(self.A) != 0)
self.potential_edges = argwhere((triu(1 - self.A) - diag(ones(self.N)) != 0))
```

Note that `self.M` is just its definition as defined in the Documentation. The other variables, however, are a bit more complex. Note that `self.edges` represents all of the places where there's an edge. Namely, `triu(self.A) != 0` gives a Boolean array of all the locations in the upper triangular where the values are not zero. This is because we define the lack of an edge as 0, whereas the existence of an edge can be any value except for 0. So, `argwhere` gives us the location of these.

As for `self.potential_edges`, this applies only for unweighted and undirected edges. Basically, if we invert `self.A`, that is to say `1 - self.A`, this will show us all the locations where edges can be placed. We subtract a diagonal of ones again, since `triu(1 - self.A)` will have ones in its diagonal. Then, we apply the same rhetoric as finding `self.edges`. As such, all of the `init` variables have been explained.

plot_graph

The purpose of this function is to make a spy plot of the adjacency matrix. Since all of these are weighted the same, 1 represents a black dot and 0 represents a white dot, on the spy plot. We first create a figure object

```
fig, ax = plt.subplots(1, 1, figsize = figsize)
```

which allows us to modify the figure and axes separately. The `1, 1` means that there is only 1 subplot. Of course, `figsize` is just the figure size. Then we can plot the actual spy via

```
ax.spy(self.A)

ax.set_ylabel('node ID, $y$', fontsize = 12)
ax.set_xlabel('node ID, $x$', fontsize = 12)
ax.xaxis.set_label_coords(0.5, 1.175)

fig.tight_layout()
```

The `spy` function generates the image of the graph, the `set_ylabel` adds text to the y-axis, and `set_xlabel` to that of the x-axis. The first parameter there is the text to be added and `fontsize` is its font size. We have to adjust the x-axis label coordinates slightly by `ax.axis.set_label_coords` so that they appear above the x-axis numbering, as opposed to below. This isn't specified for any graph, just for a `figsize = (4, 4)`, unfortunately. This will be adjusted later.

Note that `fig.tight_layout()` optimizes the shape of the image, especially for subplots with more than 1. We return `fig` and `ax` so that they can be modified if needed.

plot_networkx

Now, we can create a `networkx` plot, which is really just a fancier `matplotlib` plot. We begin the same

```
fig, ax = plt.subplots(1, 1, figsize = figsize)
```

```
G = nx.Graph()
```

However, `nx.Graph()` creates a graph object so that we can add our edges and nodes. This is done by

```
G.add_nodes_from(arange(self.N))
G.add_edges_from(self.edges)
```

such that we can graph it by

```
pos = nx.spring_layout(G)
```

```
nx.draw_networkx_nodes(G, pos = pos, node_color = node_color, node_size = node_size, alpha = node_alpha
nx.draw_networkx_edges(G, pos = pos, edge_color = edge_color, alpha = edge_alpha, ax = ax)
```

where `pos` is the position of the graph according to the `spring_layout` function. This basically optimizes the placement of the nodes and edges. We then draw the edges and nodes separately, passing in their specific parameters. This is elaborated upon more in the docstrings. However, we pass `ax = ax`, so it knows that the axis by which we're working under is the one we initially declared.

We finish off by setting

```
plt.axis('off')
```

which removes the lines that form the axes.

rewire_graph()

Lastly, we introduce notion to rewire the graph. Rewiring in particular is explained in the Documentation, but it's basically relocating edges of the network without creating any new ones. We begin by picking an edge to remove at random:

```
idx = random.randint(self.edges.shape[0])
```

The `random.randint` picks an integer at random on the domain $[0, M-1]$. We can then grab the value of that edge before removing it such that

```
r_ij, r_ji = self.edges[idx]
A_ij, A_ji = self.A[r_ij, r_ji].copy(), self.A[r_ji, r_ij].copy()
```

where `r_ij` is the edge to remove on the upper triangular and `r_ji` for the lower triangular. We then insert this into the array to retrieve the value. Normally, we can just set this to be one, but we've generalized it for weighted edges. Of course, we then set the removed edge to be zero:

```
self.A[r_ij, r_ji] = 0
self.A[r_ji, r_ij] = 0
```

Now, since this is an area where a potential edge exists, we must add this to that list and remove the edge from the actual edges:

```
self.potential_edges = append(self.potential_edges, self.edges[idx].reshape(1, 2), axis = 0)
self.edges = delete(self.edges, idx, axis = 0)
```

The `append` function creates a new array, so we just assign that to the original array we're appending on, since it takes in `self.potential_edges` as the array to be expanded on. We just have to `reshape` that particular edge because indexing it like that messes with the array structure.

Regardless, `delete` removes the value at that `idx` in `self.edges` and returns a separate array. Similar to `append`, we just place this into the original `self.edges`.

The process to add an edge is similar, but different:

```
potential_idx = random.randint(self.potential_edges.shape[0])

a_ij, a_ji = self.potential_edges[potential_idx]
```

We again get an index from the potential edges P on the range $[0, P-1]$ and then retrieve that location. Instead of saving that value, since we know that it will be zero always, we just reassign it such that

```
self.A[a_ij, a_ji] = A_ij
self.A[a_ji, a_ij] = A_ji
```

This way, if the edge is weighted, we preserve that status. Otherwise, the process of adding the new edge to the edges list and removing it from the potential edges is fairly similar:

```
self.edges = append(self.edges, self.potential_edges[potential_idx].reshape(1, 2), axis = 0)
self.potential_edges = delete(self.potential_edges, potential_idx, axis = 0)
```

Since the rewiring has finished, we have to update the edge total and the way to do this is as simple as

```
self.M = self.edges.shape[0]
```

Note that this function returns nothing, since it really only modifies the state of the graph. If desired, the modified indices could be removed!

unit testing

. . . .

TO BE CONTINUED. . . .