

Spiders like Onions: on the Network of Tor Hidden Services

Supporting material: data and code

1 Data

For each graph the Graph Builder module creates 2 files: a `services.edges` file and a `services.nodes` file. The file `services.edges` contains the edge list, and its content looks like the following:

source_ID	target_ID
2	3
4	5
5	8
6	9

The file `services.nodes` contains the node list, which is used to associate the `node_ID` to the corresponding url, and its content looks like the following:

0	23bteufi2kcqza2l.onion
1	23tjl3xpt5btiqlms.onion
2	25ffhnaechrbzwf3.onion
3	2i7aalqdpiuw36nu.onion

2 Code

Our analysis is implemented in python and it mostly relies on the `igraph` graph analysis toolbox. However, we defined a few custom methods organized into two classes and one module:

- The `aNodes` class is meant for reading and writing graph information, for drawing and plotting basic statistics.
- The `graphmanager` class is meant for managing a set of `igraph` graphs, bypassing some of `igraph`'s flaws, especially in terms of poor documentation, limited functionalities and restricted flexibility.
- The `datamanager` module is a container of functions for data preprocessing and fitting.

In short, we used `aNodes` objects to easily generating an `igraph` object from our graph file format that does not strictly adhere to any of the formats supported by `igraph`. In particular, `aNodes` comes with a `iGraphObject` method that translates an `aNodes` object into an `igraph` object. Once the `igraph` object is created, other options might be applied directly relying on `igraph` builtin methods. However, using a `graphmanager` object allowed us to apply a few metrics that `igraph` does not implement, to directly compare multiple graphs, and to obtain clear plots. For some of the plots, preprocessing and fitting data was needed, and we grouped these functions in the `datamanager` module.

2.1 The `aNodes` class

The `aNodes` class is meant to represent and analyze graphs in which nodes and/or edges may come with a set of attributes. The `aNodes` class models a graph through Python built-in data structures. In particular:

- Information about the vertices of the graph is stored in a nested dictionary called `nodedict`, whose keys are unique IDs, assumed to be of type `int`. The value associated to each node ID is a dictionary whose keys are attribute names and whose values are the content of those attributes for that node.
- Information about the edges of the graph is stored in a nested list called `edges`, whose elements are 3-tuples composed of the source node, the target node and a list of attributes of the edge.
- Two lists, respectively called `node_attributes` and `edges_attributes`, are used to keep track of the name of all attributes of the graph.

An `aNodes` object can be easily built from the `services.type.edges` and `services.nodes` files. In particular, we recommend to first use the classmethod `getEdges` to read the edge list from `services.type.edges`. The syntax for calling this method is:

```
graph = aNodes.getEdges(edges_file, edges_attributes, nodes_key)
```

`getEdges` returns an `aNodes` object in which the nodes have no attributes, whereas the edges have the attributes specified in the `edges_attributes` list. These attributes are expected to be found in the columns 3 *et seq.* of the file `edges_file`, with the first 2 columns containing the source and target of each edge. The optional `nodes_key` parameter specifies the name to be given to the node ID. Once the graph has been created based on the edge list, vertex attributes can be read from file using the method `addAttributesFromFile`. The syntax for this method is:

```
graph.addAttributesFromFile(nodes_file, node_attributes)
```

where `node_attributes` is a list of attribute names and these attributes are expected to be found in the columns 2 *et seq.* of the file `nodes_file`, with the first column containing the node ID.

The class has a wide range of features that is beyond the scope of our analysis of the Tor web graph and is not treated here. More information is available through the class' `help`.

2.2 The graphmanager class

The `graphmanager` class is a class for concurrently managing a set of `igraph` graphs. It was designed to increase the usability of `igraph` by overcoming a few issues related to poor documentation and limited functionalities.

The class is based on a nested dictionary `gdict` which is indexed by the graph names and contains all available information about those graphs. The information which can be stored in `gdict` range from the data paths (*e.g.*, for the edge file), to the whole `igraph` object, up to the results of applying a set of metrics to the graph. The class comes with methods for reading/loading and writing/saving, for computing global and local measurements, and for plotting. Let us briefly describe the most important of these methods:

read_graph A method for creating an `igraph` object from data files adhering to the formats described in Section ?? . It reads the edge list and the node list and constructs an `aNodes` object. It then generates an `igraph` object to be stored in the `gdict` dictionary. If needed, the graph is flattened (*i.e.*, self-loops and multiple edges are removed) and the giant weakly connected component is extracted.

get_intersection A method for finding the intersection of any subset of the graphs under analysis. Any node attribute can be chosen to identify matching vertices in different graphs. Only vertices and edges that exist in all selected graphs are kept. The intersection graph is added to the `gdict` dictionary.

get_metrics A method for computing all metrics considered in our paper: a comprehensive set of global metrics and four centralities (in-degree, out-degree, pagerank and betweenness). Remarkably, the considered global metrics include two measures of transitivity for directed graphs and the in- and out- global efficiency that are not available in `igraph` and are implemented as class methods in `graphmanager`. All computed metrics are stored in the `gdict` dictionary for later use.

global_metrics_table A method for dumping all global metrics into a fancy latex table. Latex formatted labels for both the graphs and the metrics can be specified.

plot_pairwise_centralities A method for plotting the pairwise comparison of any subset of the supported centrality metrics. It produces a scatter plot for each of the indicated graphs and groups these plots together. In each plot, especially central vertices are highlighted and annotated.

plot_centrality_distribution A method for plotting the distribution of any subset of the supported centrality metrics. It produces a plot for each of the indicated graphs and groups these plots together. The fibonacci binning and/or the power-law fit of the distribution can be included in the plot for any selection of metrics.

plot_pie_diagram A method for plotting a pie diagram that shows how the union of the vertex sets of all considered graphs is partitioned by taking all possible intersections. Any node attribute can be chosen to identify matching vertices in different graphs.

To show the syntax of the aforementioned methods we propose a sample pipeline. Let **names** be the names of the graphs and **basepath** be the path of the folder where all data are stored. First, we manually populate the **graphs** dictionary which will be the basis for the **gdict** dictionary:

```
graphs = name: for name in names
for name in names:
    graphs[name]['basename'] = basepath+name+'.services'
    graphs[name]['edges_file'] = graphs[name]['basename']+'.type.edges'
    graphs[name]['nodes_file'] = graphs[name]['basename']+'.nodes'
    graphs[name]['pickle_file'] = graphs[name]['basename']+'.igraph.pickle'
```

Now, let **vkey**, **vattr** and **eattr** be the vertex key, the vertex attributes and the edge attributes, respectively. We instantiate the graphmanager with:

```
gm = graphmanager(graphs, vkey, vattr, eattr)
```

We read all graphs:

```
gm.read_all_graphs()
```

We find the intersection of all graphs and add it to the **gdict**:

```
gm.get_intersection()
```

We compute all metrics:

```
gm.get_metrics()
```

We print the global metrics to table specifying latex formatted labels for both the graphs (**latex_names**) and the metrics (**metrics_names**):

```
gm.global_metrics_table(latex_names=latex_names,
metrics_names=metrics_names)
```

We plot the pie diagram indicating the labels to be used for the graphs in the plots (**plot_names**):

```
gm.plot_pie_diagram(plot_names=plot_names)
```

We plot the pairwise comparison of all supported centrality metrics indicating the labels to be used in the plots for both the graphs (**plot_names**) and the centralities (**centralities_names**):

```
gm.plot_pairwise_centralities(plot_names=plot_names,  
centralities_names=centralities_names)
```

Finally, we plot the distribution of all supported centralities indicating, other than the aforementioned labels, that fibonacci binning and power-law fit should be included for the in-degree and the pagerank centralities only:

```
gm.plot_centrality_distribution(plot_names=plot_names,  
centralities_names=centralities_names, fibonacci_binning=['indegree', 'pagerank'],  
powerlaw_fit=['indegree', 'pagerank'])
```