Complex Networks - TD1

Remy Cazabet, Lorenza Pacini

2019

1 Graph manipulation - The hard way

In this problem set we will be dealing with a graph $G = (V, E)$ where V is a set of vertices and E a set of edges. In graph theory, the adjacency list representation of a graph is a collection of unordered lists, one for each vertex in the graph. Since real world networks are often large and sparse, this is much more efficient than an adjacency matrix representation. In our python library, we define our graph with a dictionary, where node indices are keys, and values are the corresponding neighbour sets.

```
 graph = {
   "a" : {"c", "d", "g"},
   "b" : {"c", "f"},
   "c" : {"a", "b", "d", "f"},
   "d" : {"a", "c", "e", "g"},
   "e" : {"d"},
   "f" : {"b", "c"},
   "g" : {"a", "d"}
}
```

Listing 1: Python example

The aim of this problem set is to develop a rudimentary graph library, and to code from scratch some non-trivial network properties. The skeleton of the library is provided in the complementary material, leaving you room to develop the class Graph through solving the problems below. The following code is provided as a .py file.

Start by adding the following methods to the Graph class:

1. edges : return the edges of the graph
2. add vertex : add a vertex to the graph G
3. add edge : add an edge to the graph G

2 Simple function

We will now add some functions allowing to describe roughly a network

1. Add a function *vertexdegree* that output the degrees of every node, in the format of your choice
2. Add a function *degrees* that output the list of degrees of all nodes (and then plot the degree distribution, using your favorite plotting library, for instance seaborn `https://seaborn.pydata.org/tutorial/distributions.html`)
3. Add a function *density* to compute the network density.

3 Getting serious

We will now add some more complex capabilities

1. Add a function *shortest path*, that computes the shortest path between two provided nodes. Do not worry about computational time, yet.
2. Add a function *diameter*, that computes the diameter. Do not worry about computation time, yet.

4 Importation

Analyzing graphs with 7 nodes is fun, but there is even more fun: analyzing real graphs. To do that, we will need to import graphs, and therefore to write a function to do so. There are several collections of datasets on the web, but this one has the advantage of having all of them in a standard format: `http://konect.uni-koblenz.de/networks/`. Choose some graphs of moderate size, and write a function to import those networks in your library.

5 Scalability

Try to compute the diameter of a network with a few hundred nodes. What happens ? Let's try to solve this problem. Go to `https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`. Try to implement this method on your library. Compare the speed with your original implementation.

6 Bonus question

You think it is a fun problem? You're not the only one. Search for something like *fast computation diameter network* in Google, choose a fun method (exact or approximate), and implement it ! Compete with your friends for the computation on the largest graph :)

References