Lecture 4: Basics of analysis, path lengths, and small or large world networks

Eduardo López

February 9, 2017

After having studied some basic concepts of networks, it is now useful to start to do some analysis. We begin by focusing on one of the small social networks studied by Moreno, and apply some of the ideas we learned in previous chapters. This analysis will be relatively limited, and we will then proceed with our discussion of network concepts, including paths and path lengths in networks, and then connecting those concepts with some categorization of networks.

1 Analysis of Moreno's 4th grade network

1.1 Reading in the data

To perform analysis, a typical starting point is to read or import the data into the computational environment we are using. In the case of python, we need to learn a few commands to do this. To begin, it is convenient to place the data in the same folder (directory) in which your python session is running. I will assume you have done this.

Assume that the name of the dataset for Moreno's 4th grade network is 'moreno-4th-simple.dat'. It is important to know what the format of the data is. For now, I will just show a couple of lines from the file so that this is taken care of:

```
BA AB
BA BR1
DE EP
DE RY
DE YL
...
```

This corresponds to a link list format, i.e., each line in this file contains the names of a pair of nodes in the network.

In order to explain how to read the file into a python session, I will write a less compact algorithm, explaining the different parts. Later on, I shall write more compact algorithms. We begin with opening the relevant file and storing its lines into a python list. This is done with the following code:

```
In [1]: input=open('moreno-4th-simple.dat','r')
In [2]: link_list=[]
In [3]: for line in input:
   ...:     link_list.append(line.strip())
   ...:
In [4]:
```

The first line uses the open command, which takes as arguments the name of the file and the way in which python should open the file ('r': to read only, 'w': to read and write eliminating anything that was in the file already, or 'a': to attach to the file). Also, this line associates the open file with the variable 'input'. The next line initializes a list called 'link_list' which will be used to store the lines contained in the file. Then, we perform a loop on the file. python has a clever way to understand a file: it breaks it up into something very similar to a list, where each line is an element of that list. Therefore, 'for line in input:' reads the file line by line. Inside the loop, every line that is read is appended to the end of 'link_list' after having applied the string method 'strip()', used on each line to erase from the variable 'line' any leading and/or trailing invisible string characters such as newlines that may create problems later.

1.2 Preliminary data checks

Now, elements of 'link_list' still are not formatted in a way that allows us to convert them into a network. To illustrate this, see what happens when we print out some of the elements of 'link_list':

```
In [4]: link_list[:4]
Out[4]: ['BA AB', 'BA BR1', 'DE EP', 'DE RY']
```

Each of the elements of the list is a line of the file 'moreno-4th-simple.dat'. Therefore, we need to take each of those elements and break them into their component nodes to form links. We can do this in a few different ways. However, since we are already practiced in using networkx, we can proceed to create a graph with this information. Note that we have used another neat feature of python syntax, which is to indicate the indices of a list with a range; in this case '[:4]', which tells python to extract the first four elements of the list.

But before we do this, I posit that it is critical that we explore the file a bit more, slowly building up a protocol to understand the data and avoid possible unforeseen problems or inconsistencies. For the purposes of this chapter, I have simplified the actual data from Moreno a bit so that our tests are the simplest ones, but as the course progresses, we will tackle more difficult complications ('curve balls' for our baseball fans).

Let us determine if all the rows of the file have two elements, i.e., represent a link. For this, we will make use of another string method of python called '.split'. Without an argument, it scans through a string looking for blank spaces that could be 'space' or 'tab', and separates the string into substrings that do not contain blanks. Therefore

```
In [5]: lc=0
In [6]: for link in link_list:
```

```
   ...:     lc=lc+1
   ...:     LS=link.split()
   ...:     if len(LS)!=2:
   ...:         print lc
   ...:
```

The variable 'lc' counts the lines in the file, so that if we find a line violating the rules, it prints the line. The loop takes each element in 'link_list' and splits it via 'LS=link.split()', and then tests if the list 'LS' created has a length other than 2, using the Boolean operator '!='. The result comes through without printing anything, which means that indeed all the lines have two strings separated by a blank.

Another test we perform is making sure that none of the lines include self-links, which would come in the form of a line for which both strings are equal to each other. Thus, we write

```
In [7]: lc=0
In [8]: for link in link_list:
   ...:     lc=lc+1
   ...:     LS=link.split()
   ...:     if LS[0]==LS[1]:
   ...:         print lc
   ...:
```

Once again, the loop returns nothing, therefore there are no self-loops.

As a very short test, we also take note of the number of lines in the file

```
In [9]: lc
Out[9]: 45
```

If there are no repeated pairs (which can happen), this number should indicate the number of links.

Since our assumption is that each line contains a link, and this means that the first and second strings of a line are names of nodes, then we can count the number of distinct nodes in the file. This number should then match what we discover when we apply networkx (the same as with the number of links above). Therefore, we execute

```
In [10]: nl=[]
In [11]: for link in link_list:
   ....:     LS=link.split()
   ....:     if LS[0] not in nl:
   ....:         nl.append(LS[0])
   ....:     if LS[1] not in nl:
   ....:         nl.append(LS[1])
   ....:
In [12]: len(nl)
Out[13]: 31
```

The if statements make use of more python Boolean operations. So 'if not in nl:' makes use of the operator 'not', which flits a 'True' to a 'False' and vice versa. So, 'LS[0] not in nl' checks whether 'LS[0]' is already inside the list 'nl'. If it is not, then it adds it to 'nl'. The same for the next 'if'. Once the loop finished, we can check how many unique elements 'nl' has.

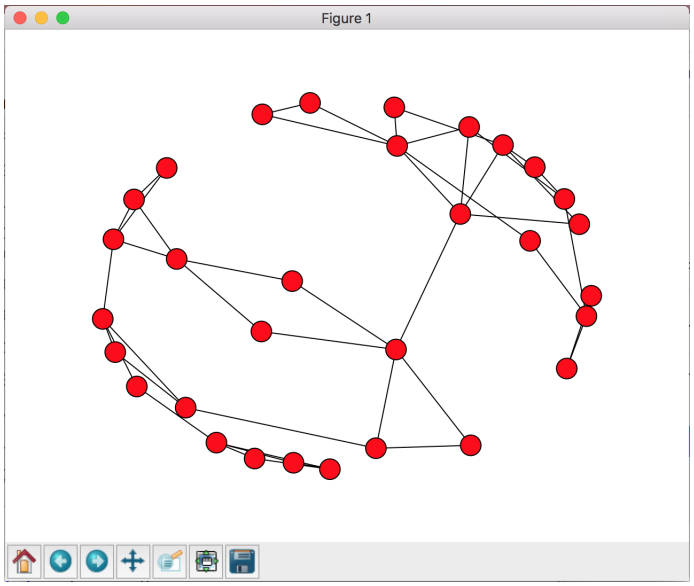

Figure 1: Moreno's simplified 4th grade network.

For now, we will be satisfied that this has checked the file for us. However, soon we will also test: 1) whether the graph is directed or not, and 2) whether the graph is connected or not. The dataset 'moreno-4th-simple.dat' has been curated to know this is not the case.

1.3 Deploying networkx and doing some analysis

We are now going to create a graph object for the data. This is done as follows:

```
In [14]: import networkx as nx
In [15]: G=nx.Graph()
In [16]: for link in link_list:
   ....:     LS=link.split()
   ....:     G.add_edge(LS[0],LS[1])
   ....:
In [17]: len(G.nodes())
Out[17]: 31
In [18]: len(G.edges())
Out[18]: 45
```

The last commands check the number of nodes and edges in the graph, and as we see, they coincide with the assessment we made above. Good news! A representation of the network can be found in Fig. 1.

We have already learned that $n = 31$ and $m = 45$. With the definition of network density, we find that

$$\rho^{(\text{Moreno 4th simplified})} \approx 0.097, \tag{1}$$

or in other words about 10% density.

Let us now construct the histogram of the network degrees. We use the lessons learned in the previous lecture for this. The code is

```
In [19]: h={}
In [20]: dsum=0
In [21]: for node in G.nodes():
    ....:     k=G.degree(node)
    ....:     dsum=dsum+k
    ....:     h[k]=h.get(k,0)+1
    ....:
In [22]: dsum
Out[22]: 90
```

We have snuck in another check, connected with the handshaking lemma, by introducing the sum 'dsum'; unsurprisingly, 'dsum=90' which is twice $m = 45$ as expected. To create the histogram of degrees, we use a new method that applies to dictionaries: we use '.get(x,y)' which has two arguments, the first is a check on whether the 'x' has been defined for the dictionary and if it has it returns the dictionary evaluated at x; if it hasn't, it returns the value y. Thus, if 'k' has been defined for 'h' then 'h.get(k,0)' returns 'h[k]', and if not it returns 0. Therefore, 'h.get(k,0)+1' is equal to 'h[k]+1' if 'k' has been defined already, or '0+1=1' if it hasn't.

We now plot this histogram with the commands we explained in the previous chapter. The code corresponds to

```
In [23]: import matplotlib.pyplot as plt
In [24]: plt.xlabel('k')
Out[24]: <matplotlib.text.Text at 0x1064b6390>
In [25]: plt.ylabel('h(k)')
Out[25]: <matplotlib.text.Text at 0x1064ced50>
In [26]: plt.bar(h.keys(),h.values())
Out[26]: <Container object of 5 artists>
In [27]: plt.show()
```

with the result shown in Fig. 2.

The majority of members of the network have a low degree. The minimum number is 2. On the other hand, there is one member of the network with degree 6. Can we tease out this node? We can do it by running

```
In [28]: khub=0
In [29]: for node in G.nodes():
    ....:     k=G.degree(node)
    ....:     if k>khub:
    ....:         khub=k
    ....:         hubnode=node
    ....:
In [30]: khub,hubnode
Out[30]: (6, 'VN')
```

This code, strictly speaking finds the 'first' node with degree equal to the largest degree in the network (if there is another node with the same degree, this code does not pick it up). In our case, we know this is unique because we can compare it to the histogram. We could in fact say that having seen the histogram first,

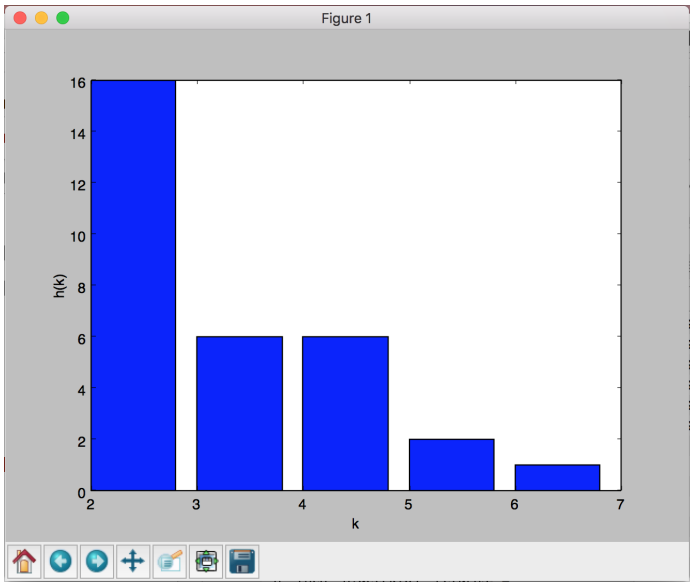

Figure 2: Histogram of degrees in Moreno's simplified 4th grade network.

we could choose to use the code above *knowing* we're gonna get the right answer. If we did not have this information, we should be more careful.

So there we are. We know some interesting things about the network: number of nodes, of links, smallest and largest degree values, and the identity of the largest degree node. Is this all we need to know? Certainly not, and this is specially true when the network to be analyzed is a lot larger than this one. But we still do not have enough tools at our disposal to perform more analysis. So we now get back to that.

2 Paths, path length, and being connected

As I mentioned earlier in the course, paths are quite meaningful for a network. Therefore, their study is critical. So, how do we define a path?

A network path is an ordered sequence of links such that consecutive links have at least one node in common. For example, a two link path can be represented by the sequence

$$((i,g),(g,j)): \quad \text{A length 2 path example,}$$

and the length of the path is simply given by the number of links that belong to the path. Note that in terms of the adjacency matrix, if we take the product $a_{ig}a_{gj}$, this is equal to 1. In other words

$$\text{if } a_{ig}a_{gj}=1 \quad \Longleftrightarrow \quad ((i,g),(g,j)): \quad \text{is length 2 path.}$$

The double direction arrow means the inverse statement is also true: in other words if the path $((i,g),(g,j))$ is there, then $a_{ig}a_{gj} = 1$.

A path has an origin and a destination. For example in the path above, the origin is i and the destination j. However, in an undirected graph, where there is no direction to the links, which node is the origin and which the destination is a matter of convention, because the path can be used in both directions. Generally speaking, we will take it as a rule that when we specify a path as we have done above, the isolated node in the first link (from left to right) is the origin, and the isolated node in the last link is the destination.

General, arbitrary paths can have peculiar features. They can go over themselves, do loops over some links, come back over a particular node and continue, etc. Here however, most of our interest is focused on a subset of paths: non-overlapping paths. These paths do not visit the same node or link twice, with the exception of the origin and destination being the same, which would form something like a necklace. Even further, within non-overlapping paths, we will spend most of our attention focused on so called *shortest paths*, i.e., the paths from a given origin and destination nodes that require the least number of links to reach from one to the other. The length of this path is usually represented by ℓ and also receives the name *hop count*. Thus, symbolically

$$\ell_{ij}: \quad \text{shortest path between nodes } i \text{ and } j. \tag{2}$$

We will soon discover that another very important property of a network is the histogram (distribution) of path lengths. This property allows us to describe how easily accessible any pair of nodes is from one another.

To conclude this section, we define what it means for two nodes to be connected. We say that

two nodes are connected if there is a path between them.

We extend this definition to a group of nodes by saying that

any group of nodes q is connected if each pair of nodes of the group is connected.

We can guarantee in undirected networks that this last property is satisfied if, out of q nodes in the group one chooses one of those nodes, and verifies that the other $q-1$ nodes are connected to the chosen node. It turns out that all nodes of the connected group can be chosen with the same result.

3 A bit about matrix algebra with regards to paths

In a supplementary section, I explain the rules for multiplication of matrices. When these rules are applied to an adjacency matrix, we find that the product provides a very interesting summary of paths between network nodes. Let us see this in action by writing one of the elements of the product of the adjacency

matrix with itself, say element ij, which yields

$$\mathbf{A}^2_{ij} = \sum_{g=1}^{n} a_{ig} a_{gj}. \tag{3}$$

Now, remember above when we wrote down what constitutes a two-link path. We said that there is a path $((i,g),(g,j))$ of length two when $a_{ig}a_{gj} = 1$. Therefore, Eq. (3) counts the number of paths of length 2 with origin i and destination j. **Note:** It is important to note that these paths need not be non-overlapping.

In general, if $\mathbf{A}$ is multiplied with itself ℓ times,

$$\mathbf{A}^{\ell}_{ij} : \text{ the number of paths of length } \ell \text{ between } i \text{ and } j. \tag{4}$$

Because in these products we count both overlapping and non-overlapping paths, we cannot use matrix products to determine the shortest path between two nodes. However, later in the course when we discuss random walks on a graph, we will come back to this matrix.

4 Small and large worlds, and Milgram's experiment

Now, as mentioned in Sec. 2, path length allows us to understand the global structure of a network better. As it turns out, path length is one of the most important distinguishing features between types of networks because it tells us a lot about the usefulness of a network.

In a general sense, there are two *broad* types of networks out there. They are small world and large world networks [1]. The big difference between these two categories is that, if we call $\langle \ell \rangle$ the average distance between node pairs in a network, then

$$\langle \ell \rangle \propto \begin{cases} n^{\alpha}, & \text{large world} \\ \log n, & \text{small world} \end{cases} \tag{5}$$

where $0 < \alpha \leq 1$. In other words, large worlds have distances that are a polynomial function of the number of nodes n, and small worlds are a logarithmic (or smaller) function of n. The explanation to this will come later, but for now, let us just give some examples in Tab. 1 of what turns out to be a small world, and what turns out to be a large world.

Without trying to get into too many details, the usual reason why networks may be either large of small relates to the idea of a space in which the network lives. This rather cryptic comment will be explained in much greater detail later. However, just think of the examples from the table: large world examples are restricted to a 2- or 3-dimensional space, whereas things such as the Internet

[1] There is in fact a third kind called ultra-small world, but for the time being let us imagine these are part of the small world category; more later

World type	Examples
Small world	Internet, some social networks, WWW, airline network
Large world	Road network, a crystal lattice

Table 1: Examples of small and large world networks

or social networks do not feel such restrictions. By the way, when a network is large world because of dimensions, then $\alpha = 1/d$ where d is the dimension.

What is important to realize is the consequence of a small or a large world in terms of distance. Imagine that we want to estimate how large a social network comprised of the entire world population could be. For the sake of argument, let us imagine that there are 10 billion people in the world, or 10^{10} people (this is a little large, as current population is about 7 billion). Now, because this is only going to modify our result by a constant, imagine the log in Eq. (5) is base 10. Then the average distance $\langle \ell \rangle$ between individuals in this network is $\log 10^{10} = 10$. On the other hand, imagine that the way the world population communicates is totally dependent on the 2-dimensional surface in which we live, which means that $\alpha = 1/2$. Then $\langle \ell \rangle$ in this case is given by $(10^{10})^{1/2} = 10^5$, i.e., 100,000. Comparing the two results, if we communicate through a small-world network the number of links to get a message across is about 10, but if we communicate in a large-world network (say where we can only talk to people we see face to face) the number of links needed to pass a message is about 100,000. This is a factor of ten thousand between the small and large world! Which one of these situations do you think we live in? Some authors believe it is this feature of small-worldness that makes some networks so essential.

4.1 Milgram's experiment

As mentioned in an earlier lecture, Milgram tested the idea of a small-world in social networks through an experiment. Let us cite the abstract of his article with Travers [1]:

"Arbitrarily selected individuals (N=296) in Nebraska and Boston are asked to generate acquaintance chains to a target person in Massachusetts, employing the "small-world method" (Milgram, 1967). Sixty-four chains reached the target person. Within this group the number of intermediaries between starters and targets is 5.2. Boston starting chains reach the target person with less intermediaries than those starting in Nebraska; subpopulations among the Nebraska group do not differ among themselves. The funneling of chains through sociometric "stars" is noted, with 48 per cent of chains passing through 3 persons before reaching the target. Applications of the method to studies of large scale social structure are discussed."

The fact that they find that there are on average 5 individuals that act as intermediaries between starter and target leads to this conclusion that there are chains of 6 individuals that include starter and target persons. It turns out

that statistically speaking there is a small inconsistency here because there are chains that never make it to their targets and this would mean that the average path length is indeed larger.

However, what this work did was confirm the suspicion that was already present in the social sciences, that acquaintanceship leads to small worlds. This is very important because all those things that travel in such networks would be susceptible to traveling quickly among nodes of the network: infectious diseases, critical information, etc. The phrase six degrees of separation is the colloquial reference to this very important notion.

From the perspective of Eq. (5), Milgram's results suggest very strongly that social network path distance is characterized by $\log n$.

5 Breath-first search and Dijkstra's algorithm

Finally, after the more technical material above, we conclude this lecture with the practical topic of finding distances in a network. It is worth prefacing this section by saying that this is the first sophisticated algorithm we build.

We focus on *breadth-first search* (BFS) which applies for finding distances in networks with homogeneous weights (all links have equal value, say 1); its generalization corresponds to Dijkstra's algorithm which is needed when links have arbitrary weights. In the physics community, breadth-first search is also known as the *burning algorithm*, because it was first used to simulate forest fires.

Before we develop the details of BFS, let us explain what the algorithm does. Given a *connected network* G and a starting node s, BFS identifies all the shortest distances between s and all other nodes of G. The way it does this is by creating an expanding front that begins at s and incorporates entire equidistant-from-s shells of nodes; it continues this until it has found all nodes of G. Each new shell identified corresponds to the nodes that are at the same distance from s, and are one more unit of distance away from s than the previous shell identified. If networks were embedded in space, BFS would identify concentric rings around s of progressively larger radii. A byproduct of BFS is the construction of a distance tree from s to all other nodes containing all the shortest paths.

Now, from the ideas above, we can see that the algorithm depends on being able to identify an active shell of nodes, a *front*, that is the part of the network most recently reached, and distinguish it from nodes that had already been visited and fully explored. To keep track of these distinct groups of nodes, we introduce the dictionaries 'visited' and 'front'; the former stores the nodes that have already been discovered by BFS and assigned their distance to s, and the later stores those nodes that have been discovered but their distance to s is not determined yet. We store distance in the dictionary 'distance'. Because some of the intricacies of the algorithm are subtle, we proceed to write it and then explain it. Here it is

```
front=[s]
```

```
d=0
visited=[s]
distance={}
distance[s]=d
while len(front)>0:
    d=d+1
    new_front=[]
    for i in front:
        for j in G.neighbors(i):
            if j in visited:
                continue
            else:
                new_front.append(j)
                distance[j]=d
                visited.append(j)
    front=new_front
```

There are three commands above to define before we explain the code. 'while (condition)' performs a loop that stops when the '(condition)' is met. The command 'continue' jumps back to the nearest loop and cycles to the next value of the loop. And the method '.neighbors(node)' operates on a graph, and it takes a node and from there creates a list neighbor nodes to our node of interest.

How does the code work? The list 'front' contains the outer most shell of the nodes that have been discovered by BFS at a given point of the execution of the code. It begins containing only 's'. Then 's' is initialized as having been visited already so it is added to list 'visited'. Also, the 'distance' dictionary is defined and 'distance[s]=0'. Next, the 'while' loop is set to run until 'front' is empty. When first entering this loop, 'front' has the node 's'. As the loop is entered, 'd' is increased by one unit because the nodes that will be found in this iteration of the loop are located one more step away from 's' than the previous step. In addition, a new list 'new_front' is created to store all the nodes that will be in the front on the next time step. Now, inside the while loop, we perform a loop over each node 'i' belonging to 'front', and within that, a loop over all the neighbor nodes 'j' of each node 'i' in 'front'. Some of those neighbors 'j' will be nodes that where already visited, and therefore need to be ignored, but other neighbors can be nodes that have never been visited, and therefore, these will be located at the new distance to 's', will be part of the 'new_front' that will be explored in the next iteration of the 'while' loop, and also become 'visited'. When all the 'i' nodes are explored to find their unvisited neighbors, 'front' is updated with 'new_front' and the 'while' loop continues if 'front' is not empty.

To wrap this up, let us do two things: 1) we define a function called BFS that receives as input 'G' and 's' and produces as output 'distances', and 2) create the histogram of distances for our Moreno dataset.

A function in python is created by this syntax

```
def funct(input1, input2, ...):
    operations
    return(output1, output2, ...)
```

and it is executed by writing

```
results=funct(input1, input2, ...)
```

and 'results' contains the tuple '(output1, output2, ...)'. With this in hand, we define:

```
def BFS(G,s):
    front=[s]
    d=0
    visited=[s]
    distance={}
    distance[s]=d
    while len(front)>0:
        d=d+1
        new_front=[]
        for i in front:
            for j in G.neighbors(i):
                if j in visited:
                    continue
                else:
                    new_front.append(j)
                    distance[j]=d
                    visited.append(j)
        front=new_front
    return(distance)
```

To create the histogram of distances for the Moreno dataset, using the largest connected node as the starting/source node, we do the following:

```
In [31]: %autoindent
Automatic indentation is: OFF
In [32]: def BFS(G,s):
   .....:     front=[s]
   .....:     d=0
   .....:     visited=[s]
   .....:     distance={}
   .....:     distance[s]=d
   .....:     while len(front)>0:
   .....:         d=d+1
   .....:         new_front=[]
   .....:         for i in front:
   .....:             for j in G.neighbors(i):
   .....:                 if j in visited:
   .....:                     continue
   .....:                 else:
   .....:                     new_front.append(j)
   .....:                     distance[j]=d
   .....:                     visited.append(j)
   .....:         front=new_front
   .....:     return(distance)
   .....:
In [33]: hl={}
In [34]: distMoreno4th=BFS(G,'VN')
In [35]: for name in distMoreno4th.keys():
   .....:     l=distMoreno4th[name]
   .....:     hl[l]=hl.get(l,0)+1
   .....:
In [36]: plt.xlabel('l')
Out[36]: <matplotlib.text.Text at 0x1083b9910>
```

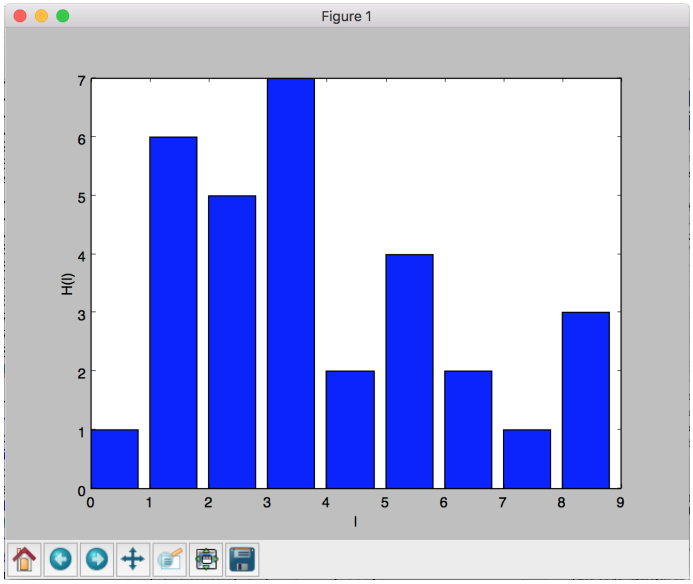

Figure 3: Histogram of distances in Moreno's simplified 4th grade network starting from the hub 'VN'.

```
In [37]: plt.ylabel('H(l)')
Out[37]: <matplotlib.text.Text at 0x10a1a92d0>
In [38]: plt.bar(hl.keys(),hl.values())
Out[38]: <Container object of 9 artists>
In [39]: plt.show()
```

and a window showing Fig. 3 is obtained.

As a closing remark, I should mention the difference between Dijkstra's algorithm and BFS: in the former, since crossing each link can have a different cost that in BFS because there are weights involved, the growing front has to be managed with much more care so that the exploration of neighbors is done in order from the node that is closest to the origin to the farthest.

We will develop our understanding of all this information in following lectures.

References

[1] J. Travers and S. Milgram, "An Experimental Study of the Small World Problem", Sociometry 32, 425–44 (1969).