3
The Internet Network

3.1 Introduction

One of the reasons behind the fact that network science has developed quickly over recent years is because of its timely description of new phenomena appearing in the present world. Indeed, the internet revolution has shaped an unprecedented society where the pervasive presence of computer-based services has changed completely the way in which we live our lives. Internet services such as the Web itself, but also Wikipedia, Facebook, and Twitter allow the exchange of information stored in servers and connected by a web of physical links. In spite of its use in a wider sense, the "Internet" is technically only the physical layer of PCs, computers, and servers connected by cables. Born to be the skeleton of a communication service between different parts of the USA in case of wartime attack, it soon became a way to connect universities and research institutes, to exchange scientific information, and later on it was exploited for its commercial uses.

The initial growth of the Internet was planned (Baran, 1964) as early as 1964, as a structure able to survive the destruction of one of its nodes, and the protocol of communications adopted (as for e-mails) was designed accordingly.

One of the most successful applications on the internet was the World Wide Web, which made possible the growth and development of a pletora of other services as, for example, Wikipedia, Facebook, Twitter, and all the other social networks.

All these various sectors are interacting with each other and reshaping their structure accordingly. Tim Berners Lee, coined the term GGG (giant global graph) to refer to the next revolution where all the information produced and stored in various services will be aggregated, categorised, and distributed in various formats according to the user's need (Berners Lee, 2007). Codes, data and/or interesting links for this chapter are available from **http://book.complexnetworks.net**.

3.2 Data from CAIDA

The Internet is the set of the various computers worldwide, connected by cables, servers etc, it indicates a physical framework. With the sentence "Looking for something on the Internet" we typically mean browsing a web site stored in one of those computers. The existence of this network is due to military research first (how to build a network for communication able to work after bombing and destruction of some of its parts), while scientific needs (how to efficiently share resource and information) appeared only late. Access to the system is not regulated by central administration and it has been made possible by non-proprietary standardized rules of communication known as the Internet

Data Science and Complex Networks. First Edition. Guido Caldarelli and Alessandro Chessa.
© Guido Caldarelli and Alessandro Chessa 2016. Published in 2016 by Oxford University Press.

Protocol Suite (TCP/IP), fixed as early as 1982. This standard has been set up in such a way as to be able to operate independently from the hardware available. Another strength of the Internet is the idea of the redundancy of connections (Baran, 1960). In the spirit of the author "A communications network that uses a moderate degree of redundancy to provide high immunity from the deleterious effects of damage of relay centres. The degree of redundancy needed is shown to be determined primarily by the amount of damage expected" This idea was successfully applied in the first planning of the Internet, where a little redundancy was present. After that, however, the structure grew with no or only limited planning and, as a result, there is no complete map of the Internet available. Nevertheless, we can use probes to determine parts of the structure. "Traceroute" is a troubleshooting application that traces the path data takes from one computer to another. While its primary aim is to detect functionality of connection, it is often used to map the Internet system. This application shows the number of hops that the data makes before reaching the host (plus extra information on how long each hop takes). These "hops" count the intermediate devices (like routers) through which data must pass between source and destination, rather than flowing directly over a simple cable. Each device along the data path constitutes a hop, or in other words is a vertex in the graph. Therefore a hop count gives the distance of two nodes in the Internet network.

In practice, by using this application (available in Linux, Mac, and Windows OS) from our Internet address, we can discover the paths connecting us to any target destination. Various projects have set up repositories of traceroute data,[1] while the most comprehensive repository is based in CAIDA (Center for Applied Internet Data Analysis).

By collecting many paths from a given address we can realise a map of how the Internet is seen by a particular observer. The result, shown in Fig. 3.1, is a map of the structure realised by the internet mapping project. At the time of writing a complete map of all connections is not available, but researchers probe constantly the portions of system that are available to obtain an as accurate as possible Internet cartography. Visualisation of the Internet as well as that of any other network is particularly important. Not by chance, in many Indo-European languages, are "to see" and "to know" are obtained from the same root *wid (producing a series of words with one or other meaning, if not both, as in Latin *video/videor*, German *wissen*, Greek (ϝ-)ιδέα, English *wise*, Serbian *vid*, Sanskrit *vid*, Czech *vidět/vědět*), not to mention the ancient Greek (ϝ-)οἶδα (I know because I have seen). To "know" about networks it is then crucial to have reliable codes to visualise them in order to disentangle their complexity. This is the case with the Internet where the system is so large that only local portions are available. Internet data from traceroute produces mostly tree-like structures whose complexity can easily be simplified using appropriate visualisation tools. The best source of data for the Internet is from the Center for Applied Internet Data Analysis (CAIDA), based at the University of California's San Diego Supercomputer Center. CAIDA is "a collaboration of different organisations in the commercial, government, and research sectors investigating practical and theoretical aspects of the Internet in order to:

[1] https://labs.ripe.net/datarepository/data-sets/iplane-traceroute-dataset

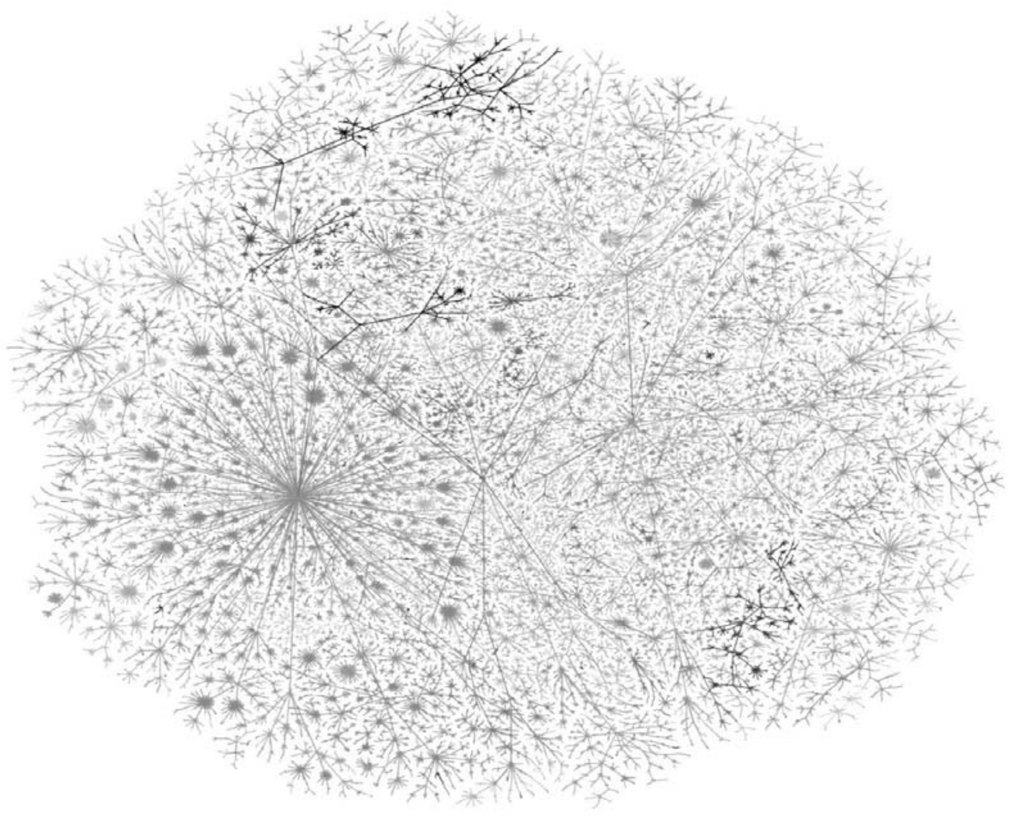

Fig. 3.1 A snaphot of the global Internet map, realised by the Internet Mapping Project (http://cheswick.com/ches/map/) on August 1998.

- provide macroscopic insights into Internet infrastructure, behavior, usage, and evolution,
- foster a collaborative environment in which data can be acquired, analyzed, and (as appropriate) shared,
- improve the integrity of the field of Internet science,
- inform science, technology, and communications public policies."

For this reason it represents a crucial place for data collection and analysis for our book

3.2.1 Visualisation

The first thing we want to show is how to visually represent a graph. Indeed, most of the important characteristics of complex networks can be spotted by just looking at them. A good visualisation algorithm, should stress the different roles of the vertices whenever it is possible to show them all. While the same graph can be drawn in a variety of forms, almost invariably the best visualisation is the one that reduces the number of crossing edges.

As in almost all cases of complex networks applications, the graphs are rather sparse, therefore, it is not particularly efficient to keep in the memory the whole adjacency matrix; rather a better choice is to consider the list of edges. For this reason, many of the available softwares for graph visualization have the list of edges as input. This is the case with the Pajek software (http://pajek.imfm.si/) which is particularly user friendly. In this case the procedure is rather simple. First just put down a row listing where you list how many vertices are in the graph. Then write the command "edges list" and then the list of edges in the format, first vertex (number) second vertex (number), so that the whole file has the form

```
Vertices 143
Edgeslist*
1  3
2  4
3  6
....
```

Other and more sophisticated desktop applications, like Gephi (http://gephi.github.io/), work with files written with a similar structure. Here we use directly the capabilities of the Python language and its graphical module Matplotlib. As a base to introduce the various centrality measures we will start from a simple and relatively small network freely available in Wikipedia.[2]

Here is a simple way to visualise this graph with Matplotlib:

Network from SVG with the best node positioning

```
import networkx as nx
from BeautifulSoup import BeautifulSoup

def Graph_from_SVG(stream):

    G=nx.Graph()

    attrs = {
        "line" :  ["x1","y1","x2","y2"]
    }

    op = open(stream,"r")
    xml = op.read()

    soup = BeautifulSoup(xml)

    count=0
    node_diz={}
    pos={}
    for attr in attrs.keys():
        tmps = soup.findAll(attr)
        for t in tmps:
```

[2]http://commons.wikimedia.org/wiki/File:Graph_betweenness.svg

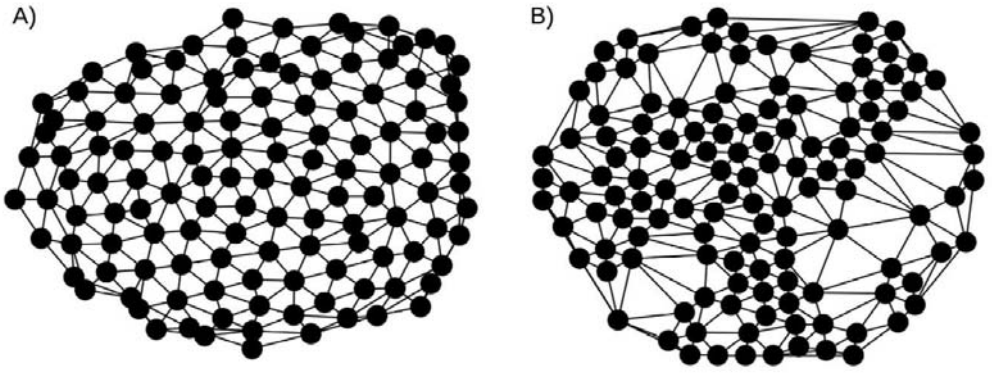

Fig. 3.2 A) The Graphviz layout for he test_graph considered in the text and B) the optimal positioning to check for node centrality measures.

```
            node1=(t['x1'],t['y1'])
            node2=(t['x2'],t['y2'])
            if not node_diz.has_key(node1):
                node_diz[node1]=str(count)
                pos[str(count)]=(float(node1[0]),float(node1[1]))
                count+=1
            if not node_diz.has_key(node2):
                node_diz[node2]=str(count)
                pos[str(count)]=(float(node2[0]),float(node2[1]))
                count+=1
            G.add_edge(node_diz[node1],node_diz[node2])
    #save the graph in an edge list format
    nx.write_edgelist(G, "./data/test_graph.dat",data=False)

    return G,pos
```

The "draw" command now presents more parameters. In particular it is possible to set the position of each node, including the "pos" parameter that is a simple Python dictionary of the type: $\{u'41' : (603.27, 84.68), \ldots\}$ where the key is the node id and the value is a couple ("tuple") formed by the two-dimensional coordinates. In the specific case we first use the graphviz layout that is able to nicely distribute nodes. Moreover the parameter "node_size" adjusts the size of the circle representing the node in order to show a more readable graph. Finally the "savefig" function stores the figure snapshot in a local file in a "png" format and with a dpi (dots per inch) equal to 200. The result is presented in Fig. 3.2A.

Even if the Graphviz layout is able to render the graph in an intelligible way, we will adopt the optimal node positions from the original SVG file with the following procedure, and at the same time we will extract from the same file its topological structure.

Visualisation tools

```
#getting the network in the SVG format
file="./data/test_graph.svg"
(G,pos)=Graph_from_SVG(file)

#plot the optimal node distribution
nx.draw(G, pos, node_size = 150, node_color='black')
#save the graph on a figure file
savefig("./data/test_network_best.png", dpi=200)

#plotting the basic network
G=nx.read_edgelist("./data/test_graph.dat")
graphviz_pos=nx.graphviz_layout(G)
nx.draw(G, graphviz_pos, node_size = 150, node_color='black')
#save the graph on a figure file
savefig("./data/test_network_graphviz.png", dpi=200)

#OUTPUT IN THE FIGURE
```

The rationale behind this algorithm is to tag the links using the coordinates as a unique id for the node names. In this way we are able both to reconstruct the topological structure of the graph from the SVG directives drawing the lines and also retrieve the exact node positions (see Fig. 3.2B for the final result).

3.3 Importance or centrality

The centrality of a vertex or edge is generally perceived as a measure of the importance of this element within the whole network. There are various ways to address these two issues and for that reason there are different measures of network centrality available. As shown in Fig. 3.3, different centrality measurements determine different sets of vertices(Perra and Fortunato, 2008; Boldi and Vigna, 2014).

3.3.1 Degree centrality

One “local” measure of centrality is to look for the vertices with the largest degrees. Indeed, being very well connected they are probably often visited by anyone travelling on the graph. This quantity called “degree centrality” is local since it can only be computed by checking the vertex itself and, in most cases, it represents a fast and reasonably accurate quantity to describe the importance of vertices in a graph. We can very quickly get the degree values for all the nodes through the following NetworkX function:

Degree sequence

```
degree_centrality=nx.degree(G)
print degree_centrality

#OUTPUT
{u'24': 7, u'25': 6, u'26': 4, u'27': 7, u'20': 7, u'21': 6,
 u'22': 4, u'23': 4, u'28': 6, u'29': 6, u'0': 5, u'4': 5,
...}
```

We then generate, plot, and save the figure (see Fig. 3.3A).

```
l=[]
res=degree_centrality
for n in G.nodes():
    if not res.has_key(n):
        res[n]=0.0
    l.append(res[n])

nx.draw_networkx_edges(G, pos)
for n in G.nodes():
    list_nodes=[n]
    color = str( (res[n]-min(l))/float((max(l)-min(l))) )
    nx.draw_networkx_nodes(G, {n:pos[n]}, [n], node_size = 100, \
    node_color =
    color)

savefig("./data/degree_200.png",dpi=200)
```

In the visualisation process we introduce new functions and paramenters. In the first place we draw just the edges with the function "draw_networkx_edges", following the precise positions through the variable "pos". Then we plot the nodes (one by one) generating a colour code proportional to the centrality, as a float between 0 and 1, normalised according to the difference between the maximum and the minimum degree centrality attained in the test graph. We will follow a similar procedure for all of the other centrality measures.

3.3.2 Closeness centrality

A non-local definition of centrality is based on the notion of distance. The quantity is non-local since we need to inspect the whole graph to compute it. The lower the distance from the other vertices the larger is the closeness. In this way we get for vertex i, the closeness c_i formula

$$c_i = \frac{1}{\sum_{j \neq i} d_{ij}}. \qquad (3.1)$$

Of course the above formula makes sense only for sites i, j in the connected component (otherwise we assume that d_{ij} is infinite). For networks that are not strongly connected, a viable alternative is harmonic centrality:

$$c_i^h = \sum_{j \neq i} \frac{1}{d_{ij}} = \sum_{d_{ij} < \infty, j \neq i} \frac{1}{d_{ij}} \qquad (3.2)$$

which replaces the implicit arithmetic mean of closeness with a harmonic mean (Boldi and Vigna, 2014). To compute these centrality measures we need a function that computes all the distances from a root node. Here we can use the BFS algorithm that we introduced in Chapter 1. This function will return the list of distances as a list of "tuples": (edge, distance):

Distance function

```
def node_distance(G,root_node):
    queue=[]
    list_distances=[]
    queue.append(root_node)
    #deleting the old keys
    if G.node[root_node].has_key('distance'):
        for n in G.nodes():
            del G.node[n]['distance']
    G.node[root_node]["distance"]=0
    while len(queue):
        working_node=queue.pop(0)
        for n in G.neighbors(working_node):
            if len(G.node[n])==0:
                G.node[n]["distance"]=G.node[working_node] \
                ["distance"]+1
                queue.append(n)
    for n in G.nodes():
        list_distances.append(((root_node,n),G.node[n]["distance"]))
    return list_distances
```

Then the closeness computes this function for all the nodes in our test network and plots it (see Fig. 3.3B).

Closeness

```
norm=0.0
diz_c={}
```

```
l_values=[]
for n in G.nodes():
    l=node_distance(G,n)
    ave_length=0
    for path in l:
        ave_length+=float(path[1])/(G.number_of_nodes()-1-0)
    norm+=1/ave_length
    diz_c[n]=1/ave_length
    l_values.append(diz_c[n])

#visualization
nx.draw_networkx_edges(G, pos)
for n in G.nodes():
    list_nodes=[n]
    color = str((diz_c[n]-min(l_values))/(max(l_values)- \
                                     min(l_values)))
    nx.draw_networkx_nodes(G, {n:pos[n]}, [n], node_size = \
                          100, node_color = color)

savefig("./data/closeness_200.png",dpi=200)
```

3.3.3 Betweenness centrality

Another "non-local" way to measure the importance of one vertex or edge is to check how often we visit it when walking on the network. When we consider all the distances among vertices in the network, we cross some "in between" sites more than once. The more often we pass through a certain site, the larger is its "betweenness" :

$$b(i) = \sum_{\substack{j,l=1,n \\ i\neq j\neq l}} \frac{\mathcal{D}_{jl}(i)}{\mathcal{D}_{jl}}, \tag{3.3}$$

where $\mathcal{D}_{jl}$ is the total number of different shortest paths (distances) going from j to l and $\mathcal{D}_{jl}(i)$ is the subset of those distances passing through i. The sum runs over all pairs with $i \neq j \neq l$. The larger the degree of a vertex, the larger is on average its betweenness; the two quantities are correlated and it is possible to connect the properties of the betweenness distribution to that of the degree distribution (Goh, Kahng and Kim, 2001; Barthélemy, 2004). See Fig. 3.3C for the result.

Betweenness

```
list_of_nodes=G.nodes()
num_of_nodes=G.number_of_nodes()
```

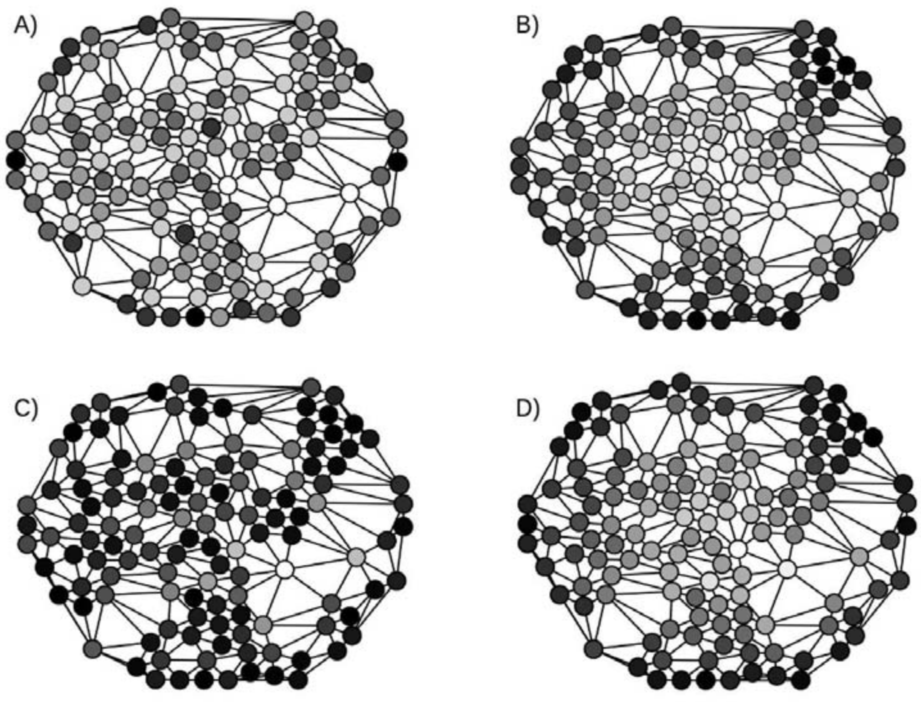

Fig. 3.3 Examples of A) degree centrality, B) closeness centrality, C) betweenness centrality, D) eigenvector centrality of the same graph.

```
bc={} #we will need this dictionary later on
for i in range(num_of_nodes-1):
    for j in range(i+1,num_of_nodes):
        paths=nx.all_shortest_paths(G,source=list_of_nodes[i], \
                                    target=list_of_nodes[j])
        count=0.0
        path_diz={}
        for p in paths:
            #print p
            count+=1.0
            for n in p[1:-1]:
                if not path_diz.has_key(n):
                    path_diz[n]=0.0
                path_diz[n]+=1.0
        for n in path_diz.keys():
            path_diz[n]=path_diz[n]/count
            if not bc.has_key(n):
                bc[n]=0.0
            bc[n]+=path_diz[n]

#visualization
```

```
l=[]
res=bc
for n in G.nodes():
    if not res.has_key(n):
        res[n]=0.0
    l.append(res[n])

nx.draw_networkx_edges(G, pos)
for n in G.nodes():
    list_nodes=[n]
    color = str( (res[n]-min(l))/(max(l)-min(l)) )
    nx.draw_networkx_nodes(G, {n:pos[n]}, [n], node_size = 100, \
                           node_color = color)

savefig("./data/betweenness_200.png",dpi=200)
```

Betweenness centrality is particularly useful in the case of community detection. Indeed, it is a measure of the "bridging" properties of one vertex/edge so that edges with large betweenness are likely to bridge different communities. Following this idea, if we remove them we can isolate the communities present in the graph (Girvan and Newman, 2002). The idea is to recursively compute the betweenness of the various edges in the network and to remove those with the largest values. In this way, isolated communities emerge from the web of connections. By iterating this procedure, the edges are removed one by one and the vertices become disconnected.

In our example algorithm, since the graph is small, we use a trivial procedure that calculates all possible minimal paths between nodes, but more refined algorithms have been introduced (Brandes, 2001) for larger graphs.

3.3.4 Eigenvector centrality

Finally we introduce a spectral centrality measure. It is based on the spectral properties of the adjacency matrix A (other measures are based on simple functions of it). The starting point is to define the centrality of a vertex i as the average of the centrality of its neighbours, i.e.

$$c_i = \frac{1}{\lambda} \sum_{j=1,N} a_{ij} c_j. \tag{3.4}$$

In its vectorial form the above equation can be written as

$$A\vec{c} = \lambda\vec{c}. \tag{3.5}$$

That is, the centrality is an eigenvector of the adjacency matrix A, where λ is the corresponding eigenvalue. To have a physical sense the above eigenvalue must be real, but in general this is not always ensured. To partly overcome these problems it is a good choice to take λ as the largest (in absolute value) eigenvalue of matrix A. As we see later (see Section 4.3), by using the Perron–Frobenius theorem, this means that if A is irreducible, or equivalently if the graph is (strongly) connected, then the eigenvector $\vec{c}$ is both unique and positive.

To solve the above problem numerically we use a power iteration method also known as the Von Mises iteration method. The idea is to start with a good approximation of the eigenvector related to the largest eigenvalue (dominant eigenvector), or directly from a random one, and iterate the vector coefficients according to the relation

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|}. \tag{3.6}$$

In this way, at every iteration, the vector b_k is multiplied by the matrix A and normalised. In order for a subsequence of (b_k) to converge, it is sufficient that the matrix A has an eigenvalue that is strictly greater in magnitude than its other eigenvalues and also the starting vector b_0 must have a nonzero component in the direction of an eigenvector associated with the dominant eigenvalue.

Eigenvector centrality

```
#networkx eigenvector centrality
centrality=nx.eigenvector_centrality_numpy(G)

#visualization
l=[]
res=centrality
for n in G.nodes():
    if not res.has_key(n):
        res[n]=0.0
    l.append(res[n])

nx.draw_networkx_edges(G, pos)
for n in G.nodes():
    list_nodes=[n]
    color = str( (res[n]-min(l))/(max(l)-min(l)) )
    nx.draw_networkx_nodes(G, {n:pos[n]}, [n], node_size = 100, \
    node_color = color)

savefig("eigenvetor_200.png",dpi=200)
```

The result of this procedure is illustrated in Fig. 3.3D.

3.4 Robustness and resilience, giant component

Robustness and resilience are concepts often invoked in the field of critical infrastructure, as for example with the Internet, water pipelines, and electricity grid. It is

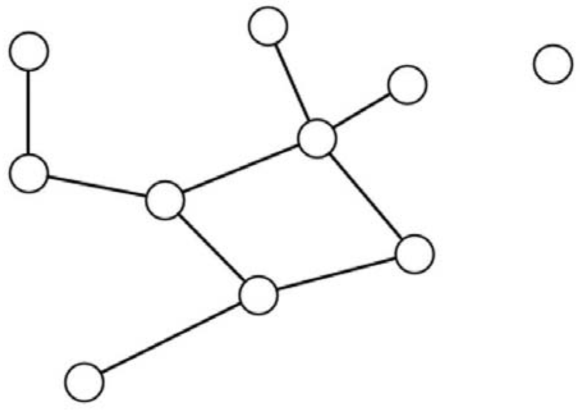
Fig. 3.4 An example of a network with two components.

important to note that they refer to distinct even if similar properties. The first quantity i.e. *robustness* iss more a static property referring to how well a system can resist an attack or failures, before being disrupted. The second quantity, i.e. *resilience* is more dynamic and describes how a system can reshape itself to avoid being disrupted.

In graph theory and network analysis, sometimes the two terms are confused and in general the "robustness" of a graph indicates the probability of remaining connected under successive removal of edges or vertices. Traditionally one of the simplest formulas for measuring robustness checks for the size of the giant connected component(Callaway, Newman, Strogatz and Watts, 2000) (i.e. the largest subset of the system which may correspond to the whole system at the beginning). When the size reduces, the network starts deteriorating. Here we present a "built-in" algorithm to compute this quantity. Networkx already has an efficient function to perform this operation, but as an example of the use of the BFS introduced previously (see Section 1.5), we code this procedure from scratch. In the first place we need a small test network with a single disconnected component to verify proper discovery of the network components (see Fig. 3.4):

Generating the graph with two components

```
G_test=nx.Graph()
G_test.add_edges_from([('A','B'),('A','C'),('C','D'),('C','E'),
                       ('D','F'), ('D','H'),('D','G'),('E','G'),
                       ('E','I')])
#disconnetted node
G_test.add_node('X')
nx.draw(G_test)

savefig("components_200.png",dpi=200)
```

Then the idea is to apply the BFS starting from a node until it discovers the component attached to it, and then progressively extract all possible components. In our case the output will be the GCC and the number of components present in the network:

Giant component through a breadth first search

```
def giant_component_size(G_input):

    G=G_input.copy()

    components=[]

    node_list=G.nodes()

    while len(node_list)!=0:
        root_node=node_list[0]
        component_list=[]
        component_list.append(root_node)
        queue=[]
        queue.append(root_node)
        G.node[root_node]["visited"]=True
        while len(queue):
            working_node=queue.pop(0)
            for n in G.neighbors(working_node):
                #check if any node attribute exists
                if len(G.node[n])==0:
                    G.node[n]["visited"]=True
                    queue.append(n)
                    component_list.append(n)
        components.append((len(component_list),component_list))
        #remove the nodes of the component just discovered
        for i in component_list: node_list.remove(i)
    components.sort(reverse=True)

    GiantComponent=components[0][1]
    SizeGiantComponent=components[0][0]

    return GiantComponent,len(components)

(GCC, num_components)=giant_component_size(G_test)
print "Giant Connected Component:",GCC
print "Number of components:",num_components

#OUTPUT
Giant Connected Component: ['A', 'C', 'B', 'E', 'D', 'I',
            'G', 'H', 'F']
```

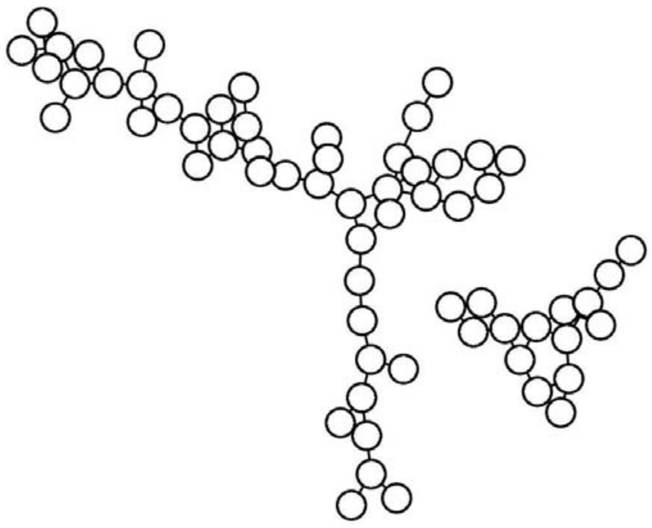

Fig. 3.5 The GCC broken into two components after the node deleting procedure.

```
Number of components: 2
```

Now we will try to break up the GCC randomly deleting some of its nodes. To check the result visually we will apply this new function to the network defined at the beginning of this chapter (Fig. 3.5):

Breaking the GCC

```
import copy

def breaking_graph(H,node_list):
    #define the new graph as the subgraph induced by the GCC
    n_l=copy.deepcopy(node_list)
    #iterate deleting from the GCC while the graph comprises
    #one component (num_components=1)
    num_components=1
    count=0
    while num_components==1:
        count+=1
        #select at random an element in the node list
        #node_to_delete=random.choice(H.nodes())
        #select a node according to the betweenness ranking
        #(the last in the list)
        node_to_delete=n_l.pop()
        H.remove_node(node_to_delete)
        #(GCC,num_components)=giant_component_size(H)
        num_components=nx.number_connected_components(H)
    return count

(GCC, num_components)=giant_component_size(G_test)

G_GCC = G_test.subgraph(GCC)

random_list=copy.deepcopy(G_GCC.nodes())
random.shuffle(random_list)

c=breaking_graph(G_GCC,random_list)

print "num of iterations:", c

graphviz_pos=nx.graphviz_layout(G_GCC)

nx.draw(G_GCC, graphviz_pos, node_size = 200, with_labels=True)

savefig("./data/broken_component_200.png",dpi=200)

#OUTPUT
num of iterations: 1
```

Using the previous code and iterating over 1000 possible realisations of the random deleting procedure, we can get the typical number of iterations to break up the network. We will test this robustness procedure on data describing a map of the Internet at the Autonomous System (AS) level (see http://www.cosinproject.eu/extra/data/internet/nlanr.html):

Breaking up the giant connected component randomly

```
#loading the Autonomous System (AS) graph
G_AS=nx.read_edgelist("./data/AS-19971108.dat")
print "number of nodes:",G_AS.number_of_nodes(), \
"number of edges:",G_AS.number_of_edges()

(GCC, num_components)=giant_component_size(G_AS)

n_iter=1000
```

```
count=0.0
for i in range(n_iter):
    G_GCC = G_AS.subgraph(GCC)
    random_list=copy.deepcopy(G_GCC.nodes())
    random.shuffle(random_list)
    c=breaking_graph(G_GCC,random_list)
    count+=c

print "average iterations to break GCC:",count/n_iter

#OUTPUT
number of nodes: 3015 number of edges: 5156
average iterations to break GCC: 8.35
```

In this case, on average, the procedure takes approximatively eight steps to break the giant component. The robustness of the network depends on both the node deleting procedure and on the nature of the network itself. We can take out nodes randomly or follow a particular order based on some centrality measure. Moreover there are networks that are intrinsically fragile because of their topological structure (mostly regardless of the procedure adopted, e.g. trees). Previously we probed our network robustness against a random node deletion. Now we try to break it following an order that is related to a particular ranking based on the betweenness centrality measure. The general result is that following the centrality measure the network breaks down much more quickly. Here we use the betweenness centrality measure from the Networkx library. The final result is that the AS network breaks immediately, just after one node deletion.

Breaking up the giant connected component with betweenness centrality

```
import operator

G_GCC = G_AS.subgraph(GCC)

node_centrality=nx.betweenness_centrality(G_GCC, k=None, \
normalized=True, weight=None, endpoints=False, seed=None)
#node_centrality=nx.degree_centrality(G)

sorted_bc = sorted(node_centrality.items(), \
key=operator.itemgetter(1))

#selecting the ranked node list
```

```
node_ranking=[]
for e in sorted_bc:
    node_ranking.append(e[0])

c=breaking_graph(G_GCC,node_ranking)

print "num of iterations:", c

#OUTPUT
num of iterations: 1
```