4

World Wide Web, Wikipedia, and Social Networks

4.1 Introduction

Among the various services working on the Internet, one of the most successful has been the World Wide Web (WWW). In spite of the difference between the two services, even now it is common to find people interchangeably using internet (the physical device) for the WWW (the set of media and documents connected by hyperlinks). The success of the WWW has been immense. It represents the largest construction made by man, it encompasses most of world knowledge, it allows the exchange of information and feelings between people (even if physically very far apart) and it has become the environment in which different projects such as Wikipedia (an on-line encyclopaedia in almost every language of the world); Facebook (initially a system to stay in touch with a "restricted" group of friend, but now also a platform for messaging, mailing, e-commerce, and a source of news and information), and other social networks (RenRen operating in China, VK operating in Russia), and also a site for microblogging such as Twitter.

The WWW bases its success on the potential offered by the hypertext markup language (html). Thanks to this language it is possible to link documents and media with each other creating a network of information. Based on this idea, Tim Berners Lee (a collaborator of the physics laboratory CERN) decided in 1989 to put together some material in a series of pages that could be "browsed" with a specific piece of software. In such a delocalised structure it then becomes necessary to specify the location where the information is stored. This is possible thanks to the protocols of address for servers on the Internet. In any case the Internet numerical structure (i.e. an address of the kind 193.67.163.111) is rather uninformative and too restricted to keep track of all documents. As a result, WWW developed a series of addresses of the form "www.oup.com". Thanks to a hierarchical classification, an almost infinite series of documents can be mapped below this address. The mapping is stored in specific servers named "domain name servers (DNS)".

From the very beginning, many institutions, companies and private individuals put their information on line, and at the same time people worldwide started building their personal and business home pages. As the number of pages started to grow it was necessary to have a "telephone list" of the information present on the system. The first and most obvious solution was to compile a topical list. Web searching was mostly done through a page of a company named Yahoo! (yet another hierarchical officious oracle !), which was an "officious" list of links organised hierarchically. This meant that any

Data Science and Complex Networks. First Edition. Guido Caldarelli and Alessandro Chessa.

new page had to be found and put manually into this artificial taxonomy to be present in the list. This task became more and more difficult as the numbers exploded (where and how to find all new pages?) and the content became more and more complex (how to assess the category of a web page?) to classify. Codes, data and/or links for this chapter are available from **http://book.complexnetworks.net**.

4.2 Data from various sources

4.2.1 WWW

The WWW is a classic example of big data. Often described as the largest coherent structure created by humans; actually its size (in the order of tens of billions) only refers to the static pages. Indeed, some web pages are created on demand (think of web pages for the days in a calendar) when users look for them, so that the real size of the WWW is virtually infinite. It is therefore of the utmost importance to be able to handle these series of data, and whenever possible to consider properly defined subsets of them. To that purpose we would suggest starting an exploration of the web from a set of databases collected by the Laboratory for web algorithmics of the University of Milan, Italy.

- At http://law.di.unimi.it we can find information on this site;
- http://law.di.unimi.it/datasets.php contains a series of data collected and stored in compressed form;
- http://webgraph.di.unimi.it/ contains information about the Webgraph compressed graph format and instructions on how to extract it.

On this site one can download crawls of the web of different sizes: "small" ones to test software (about 10^5 sites) to larger ones (about $10^6 - 10^7$ sites). The procedure for getting networks from these compressed files is not simple and for the benefit of the reader we have performed this task, generating a "ready to use" edge list, particularly related to a portion of the European domain name ".eu" (http://law.di.unimi.it/webdata/eu-2005/).

Code for loading the ".eu" portion of the WWW in 2005

```
import networkx as nx

#defining the eu directed graph
eu_DG=nx.DiGraph()
#retrieve just the portion of the first 1M edges of the .eu domain
#crawled in 2005
eu_DG=nx.read_edgelist('./data/eu-2005_1M.arcs', \
                       create_using=nx.DiGraph())
#generate the dictionary of node_is -> urls
file_urls=open('./data/eu-2005.urls')
count=0
dic_nodid_urls={}
```

```
while True:
    next_line=file_urls.readline()
    if not next_line:
        break
    next_line[:-1]
    dic_nodid_urls[str(count)]=next_line[:-1]
    count=count+1
file_urls.close()

#generate the strongly connected component
scc=[(len(c),c) for c in sorted( nx.strongly_connected_components \
                                (eu_DG), key=len, reverse=True)][0][1]
eu_DG_SCC = eu_DG.subgraph(scc)
```

4.2.2 Twitter

Twitter (twitter.com) is a microblogging platform, which is a service that allows its users to exchange short comments ("tweets"). Its rapid success has now made it possible to track down messages and their forwarding ("retweets") to millions of bloggers. In Twitter, each user has an account from which it is possible to write up to 140 characters in "tweets" to followers. Some users have tens of thousands of followers, others much fewer. Such "following" relationship is not reciprocal (i.e. if A follows B, not necessarily does B follow A). Twitter and Facebook are two clear cases where networks help in measuring social relationships. In particular they are a typical case of study of the new computational social science(Lazer *et al.*, 2009; Gonçalves *et al.*, 2011; Del Vicario *et al.*, 2016) Twitter provides application programming interfaces (APIs) to access tweets and information about tweets and users (https://dev.twitter.com/docs). The Python module for interacting with the Twitter API is Twython and can be reached from this link: https://twython.readthedocs.org/.

Code for the opening of tweets with the API

```
#To get your own KEYS and TOKENS visit the following page:
#https://dev.twitter.com/docs/auth/tokens-devtwittercom
#(you have to sign in before with your Twitter account)

from twython import Twython

APP_KEY='XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
APP_SECRET='XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
OAUTH_TOKEN='XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
OAUTH_TOKEN_SECRET='XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX'
```

```
twitter_connection=Twython(APP_KEY, APP_SECRET, \
                           OAUTH_TOKEN,OAUTH_TOKEN_SECRET)
```

From here it is now possible to get data from Twitter such as for example, the tweets from the timeline of the user.

How to get the timeline

```
#the following tweets and query results
#depend on the KEYS and TOKENS of the user

res=twitter_connection.get_home_timeline()
for t in res[:5]:
    #print the text of the first 5 tweets of the actual timeline
    print 'Text of the tweet:',t[u'text']
    #for each tweet print the mentioned users
    print 'mentions:',
    for m in t[u'entities'][u'user_mentions']:
        print m[u'screen_name'],
    print '\r'

#OUTPUT
Tweet: Vincere e vinceremo, l'ultimo scandalo del doping di
stato, non il solo eh. https://t.co/8Mzz2RmC3H
mentions:
Tweet: Ad IO SPAZIO sabato 14 novembre sar\`a ospite un amico
ed un grande professionista. @michelecucuzza
#gramigna https://t.co/wh5zdph98p
mentions: michelecucuzza
Tweet: RT @Jodinette: #LaPhraseQuiMenerve t'es contre l'euthanasie?
T'es contre le droit de mourir dans la dignit\'e!
https://t.co/0xywnSVNGU
mentions: Jodinette
Tweet: 10 days of round the clock curfew. Very harsh policy,
punishes all the people stuck there.  https://t.co/eYw1pPET02
mentions:
Tweet: Terrorismo internazionale, Merano crocevia degli aspiranti
jihadisti https://t.co/PvUP9x0chn
mentions:
```

In the following case we check information about the President of the United States, Mr. Barak Obama; in particular his location and number of followers.

How to get user information

```
res=twitter_connection.show_user(screen_name='@BarackObama')
print res
print 'location: ',res[u'location']
print 'number of followers: ',res['followers_count']

#OUTPUT
{u'follow_request_sent': False, u'has_extended_profile':
True, u'profile_use_background_image': True, u'profile_text_color':
 u'333333',
u'default_profile_image': False, u'id': 813286,
u'profile_background_image_url_https':
u'https://pbs.twimg.com/profile_background_images/
451819093436268544/kLbRvwBg.png',u'verified': True,
u'profile_location': None, u'profile_image_url_https':
u'https://pbs.twimg.com/profile_images/451007105391022080/
iu1f7brY_normal.png', u'profile_sidebar_fill_color': u'C2E0F6',
u'entities':{u'url': {u'urls':[{u'url': u'http://t.co/O5Woad92z1',
u'indices': [0, 22], u'expanded_url':
u'http://www.barackobama.com', u'display_url': u'barackobama.com'}]},
u'description': {u'urls': []}},...
...

location:  Washington, DC
number of followers:  65822369
```

After the Twitter timeline and user information one could be interested in getting a bunch of tweets related to a particular topic or hashtag. To this end the Twitter API offers a search function. In the following example we will use it to extract some tweets in which the hashtag "#ebola" appears.

Retrieving tweets with the "search" function

```
res=twitter_connection.search(q='#ebola', count=2)
for t in res['statuses']:
    print "Text of the tweet:",t[u'text']

#OUTPUT
Tweet: Read my interview on @CNN about how #SierraLeone can turn
its economy around after #Ebola https://t.co/hmeCKT5RVC
Tweet: Brazil tests man for #Ebola, puts others under observation
https://t.co/0VGSTPiqNK https://t.co/F17Xjzos09
```

Following a similar approach one could start to monitor the activity of a series of politicians in a given state and, more interestingly, the activity of all users related to these persons. Various studies have shown clearly how this information could be extremely valuable for providing an idea of the political situation of a country, especially around elections time (Eom *et al.*, 2015; Caldarelli *et al.*, 2014; Tumasjan *et al.*, 2010; DiGrazia *et al.*, 2013; Albrecht *et al.*, 2007; Adamic and Glance, 2005). An archive of some cured data for Italian political elections is available at http://www.linkalab.it/data

4.2.3 Wikipedia

Wikipedia is another service based on the WWW. It consists of a series of web pages written by a very large community of editors, on a variety of different arguments. In time it became an open access/open edit on-line encyclopaedia, whose reliability is ensured by the constant control of editors and users. The various pages are interconnected (links between existing pages are constantly being created by readers and editors) forming one of the largest thematic subnetworks of the WWW. For this reason, it has for quite a long time attracted the interest of scientists (Martin, 2011; Capocci *et al.*, 2006; Zlatić *et al.*, 2006). Interest in this subset of WWW pages is based on a series of reasons.

- Wikipedia is a well defined subgraph of the WWW; indeed it forms a thematic subset, thereby creating a natural laboratory for WWW studies.
- Over time Wikipedia has developed in different languages, so that various subsets of Wikipedia of different sizes are now available. Furthermore, Wikipedia networks allow us to test whether different cultures tend to organise web pages differently.
- All information on the Wikipedia graph is available, even its growth history, with a time stamp for any additions to the system.
- Wikipedia pages tend (where possible) to cite other Wikipedia pages, so that the whole system is contained.

In such a (extremely well connected) network it is interesting to see if the links connecting two pages (lemmas of the encyclopaedia) determine communities of concepts and ultimately define a bottom-up taxonomy of reciprocal concepts (as one would expect). For the purpose of this book it is important to note that all the data about the present shape of the network (and its growth) is publicly available and can be downloaded from

- http://dumps.wikimedia.org/

A general introduction to the subject and details of how to manage Wikipedia database files is presented on the site[1]. Dumps, varying from the largest dataset of the English version to smaller samples. We can start from the latter and then move to larger and larger datasets. In the following, for example, we shall use a small portion of Wikipedia, that "in Limba Sarda" (Sardinian), which is at the moment formed from about 4500 articles[2]. Even though the procedure of querying the Mysql database is

[1] https://meta.wikimedia.org/wiki/Data_dumps

[2] https://sc.wikipedia.org/

beyond the scope of the present book, we will sketch in the following core the main steps to extracting the hyperlink information from the dump. But in the end we will store the link structure and the page titles in a local file that we will use later on to load and to populate the proper Networkx and dictionary structures. It is possible to find the structure of the Pagelinks and Page table dumps in the following links: https://www.mediawiki.org/wiki/Manual:Pagelinks_table

https://www.mediawiki.org/wiki/Manual:Page_table

Opening the Wikipedia Sardinian dump

```
#You can skip the following cell if you don't have mysql installed
#and use directly the filesscwiki_edgelist.dat and
#scwiki_page_titles.dat you will find in the 'data' directory


#open the DB connection
#the scwiki mysql dumps scwiki-20151102-pagelinks.sql and
#scwiki-20151102-page.sql (both in the 'data' dir) have to be loaded
#in the tables "pagelinks" and "page" of the DB "scwiki_db" (to be
#created) before to launch this procedure through these commands:
#mysql -u<user> -p<password> scwiki_db< scwiki-20151102-pagelinks.sql
#mysql -u<user> -p<password> scwiki_db< scwiki-20151102-page.sql

import _mysql

scwiki_db=_mysql.connect(host="localhost",user="root", \
                         passwd="mumonkan",db="scwiki_db")

#extract the hyperlinks information with a SQL query
#from the mysql DB and storing them in a local file
scwiki_db.query("""SELECT pagelinks.pl_from, page.page_id
FROM page,pagelinks
WHERE page.page_title=pagelinks.pl_title""")
r=scwiki_db.use_result()
f=open("./data/scwiki_edgelist.dat",'w')
res=r.fetch_row()
while res!=():
    f.write(res[0][0]+" "+res[0][1]+"\n")
    res=r.fetch_row()
f.close()

#extract the title information with a SQL query
#from the mysql DB and storing them in a local file
scwiki_db.query("SELECT page.page_id,page.page_title FROM page")
```

```
r=scwiki_db.use_result()
f=open("./data/scwiki_page_titles.dat",'w')
res=r.fetch_row()
while res!=():
    f.write(res[0][0]+" "+res[0][1]+"\n")
    res=r.fetch_row()
f.close()
```

4.2.4 Wikipedia taxonomy

Since Wikipedia is a means of organising knowledge (Gonzaga *et al.*, 2001), it is interesting to check whether the structures arising from different languages and then different cultures have some sort of universality (Muchnik *et al.*, 2007; Capocci *et al.*, 2008). Furthermore the network formed by articles and hyperlinks together could provide a self-organized way to gather Wikipedia articles into categories; a classification that it is currently created upon the agreement of the whole Wikipedia community. The simplest way to create a taxonomy is by use of a tree in the shape of the Linnean taxonomy of living organisms (Linnaeus, 1735). This topic has been thoroughly studied over past years. Historically, the complexity (i.e. the fat-tailed distribution of the number of offspring at the various levels) of the structure of natural taxonomic trees from plants and animals (Willis and Yule, 1922) led to the Yule model for the growth of trees (Yule, 1925), where mutations in a population of individuals may eventually form a series of different species in the same genus.

Such a clean structure does not, unfortunately, fully apply to Wikipedia. Indeed, articles and categories will not strictly form a perfect tree, since an article or a category may happen to be the offspring of more than one parent category. For this reason the taxonomy of articles is represented in this case as a direct acyclic graph. This means that the taxonomy must be considered only as a soft partition, where the intersection between classes is different from zero. In this case one deals with (so-called) fuzzy partitions.

4.3 Bringing order to the WWW

In this section we present a short overview of the various methods that have been presented and made public to infer the importance (centrality) of pages in the WWW. Nowadays, modern search engines have (very likely) far more complicated algorithms and methods, nevertheless the original methods are still important for other cases of study and they make an excellent stage for presenting important concepts of graph theory. In the cases of studies that we present here, we define the importance of a page only topologically i.e. without entering into semantic analysis of the content of a single page. The first algorithm using such an approach was introduced in 1999 (Kleinberg, 1999) under the name HITS (Hyperlink-Induced Topic Search).

4.3.1 HITS algorithm

As a first approximation, let's make a basic differentiation of pages into two categories:

- *authorities* i.e. pages that contain relevant information (train timetable, food recipes, formulas of algebra);
- *hubs* i.e. pages that do not necessarily contain information, but (as with Yahoo! pages) have links to pages where the information is stored.

Apart from limiting cases, every page i has both an authority score $au(i)$ and a hub score $h(i)$, that are computed via a mutual recursion. In particular we define the authority of one page as proportional to the sum of the hub scores of the pages pointing to it,

$$au(i) \propto \sum_{j \to i} h(j). \tag{4.1}$$

Similarly, the hub score of one page is proportional to the authority scores of the pages reached from the hub,

$$h(i) \propto \sum_{i \to j} au(j). \tag{4.2}$$

To ensure convergence of the above recursion, a good method is to normalise the values of $h(i)$ and $a(i)$ at every iteration such that $\sum_{i=1}^{n} h(i) = \sum_{i=1}^{n} au(i) = 1$.

HITS algorithm

```
def HITS_algorithm(DG):
    auth={}
    hub={}

    k=1000 #number of steps

    for n in DG.nodes():
        auth[n]=1.0
        hub[n]=1.0

    for k in range(k):
        norm=0.0
        for n in DG.nodes():
            auth[n]=0.0
            for p in DG.predecessors(n):
                auth[n]+=hub[p]
            norm+=auth[n]**2.0
        norm=norm**0.5
        for n in DG.nodes():
            auth[n]=auth[n]/norm
```

```
        norm=0.0
        for n in DG.nodes():
            hub[n]=0.0
            for s in DG.successors(n):
                hub[n]+=auth[s]
            norm+=hub[n]**2.0
        norm=norm**0.5
        for n in DG.nodes():
            hub[n]=hub[n]/norm

        return auth,hub

DG=nx.DiGraph()

DG.add_edges_from([('A','B'),('B','C'),('A','D'), \
                   ('D','B'),('C','D'),('C','A')])

#plot the graph
nx.draw(DG,with_labels=True)

(auth,hub)=HITS_algorithm(DG)

print auth
print hub

#OUTPUT
{'A': 0.31622776601683794, 'C': 0.31622776601683794,
'B': 0.6324555320336759, 'D': 0.6324555320336759}
{'A': 0.7302967433402215, 'C': 0.5477225575051661,
'B': 0.18257418583505539, 'D': 0.3651483716701107}
```

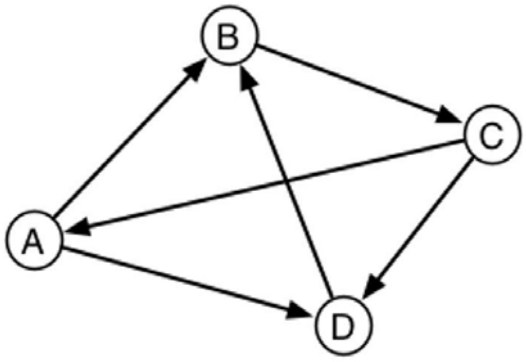
4.3.2 Spectral properties

This method can be (qualitatively, not considering the normalisation problems) described by means of linear algebra. As seen in the first chapter (see Section 1.3), a graph can be equivalently represented by means of a matrix of numbers, that is, with its adjacency matrix, as shown in the graph in Fig. 4.1.

The equation giving rise to the hub score can be written as

$$h(i) \propto \sum_{i \to j} au(j) \to h(i) \propto \sum_{j=1}^{n} a_{ij} au(j) \to \vec{h} \propto A\vec{au} \tag{4.3}$$

and similarly for the authorities we obtain:



	A	B	C	D
A	0	1	0	1
B	0	0	1	0
C	1	0	0	1
D	0	1	0	0

Fig. 4.1 A simple oriented graph with its adjacency matrix.

$$au(i) \propto \sum_{j \to i} h(j) \to au(i) \propto \sum_{j=1}^{n} a_{ij}^T h(j) \to \vec{u} \propto A^T \vec{h}, \tag{4.4}$$

where a_{ij}^T are the elements of the matrix A^T that is the transpose of A (this means that $a_{ij}^T = a_{ji}$.

How to transpose and multiply a matrix

```
def matrix_transpose(M):
    M_out=[]
    for c in range(len(M[0])):
        M_out.append([])
        for r in range(len(M)):
            M_out[c].append(M[r][c])
    return M_out


def matrix_multiplication(M1,M2):
    M_out=[]
    for r in range(len(M1)):
        M_out.append([])
        for j in range(len(M2[0])):
            e=0.0
            for i in range(len(M1[r])):
                e+=M1[r][i]*M2[i][j]
            M_out[r].append(e)
    return M_out
```

```
adjacency_matrix1=[
                   [0,1,0,1],
                   [1,0,1,1],
                   [0,1,0,0]
                   ]

adjacency_matrix2=matrix_transpose(adjacency_matrix1)

print "Transpose adjacency matrix:",adjacency_matrix2

res_mul=matrix_multiplication(adjacency_matrix1,adjacency_matrix2)

print "Matrix multiplication:",res_mul

#OUTPUT
Transpose adjacency matrix: [[0, 1, 0], [1, 0, 1], [0, 1, 0],
[1, 1, 0]]
Matrix multiplication: [[2.0, 1.0, 1.0], [1.0, 3.0, 0.0],
[1.0, 0.0, 1.0]]
```

By combining (4.3) and (4.4) we obtain

$$\begin{aligned} \vec{h} \propto AA^T\vec{h} = \lambda_h AA^T\vec{h}, \\ a\vec{u} \propto A^TA a\vec{u} = \lambda_{au} A^TA\vec{u}. \end{aligned} \tag{4.5}$$

That is an eigenvalue problem for the matrices $M \equiv AA^T$ and $M^T \equiv A^TA$.

- M (and therefore its transpose) is real and symmetric, so its eigenvalues are real;
- M is non-negative (i.e. the entries are at least 0 or larger); if we can find a $k > 0$ such that $M^k >> 0$, that is, all of the entries are strictly larger than 0, then M is *primitive*. If M is a primitive matrix:
  - the largest eigenvalue λ of M is positive and of multiplicity 1;
  - every other eigenvalue of M is in modulus strictly less than λ;
  - the largest eigenvalue λ has a corresponding eigenvector with all entries positive.

Being a primitive matrix means in physical terms that the graph defined by the adjacency matrix must have no dangling ends or sinks and that it is possible to reach any page from any starting point. In all of the above hypothesis convergence is ensured.

Principal eigenvalue/vector extraction (power iteration)

```
adjacency_matrix=[
                   [0,1,0,1],
```

```
                  [1,0,1,1],
                  [0,1,0,0],
                  [1,1,0,0]
                  ]
vector=[
        [0.21],
        [0.34],
        [0.52],
        [0.49]
        ]

for i in range(100): #100 iterations is enough for the convergence!
    res=matrix_multiplication(adjacency_matrix,vector)
    norm_sq=0.0
    for r in res:
        norm_sq=norm_sq+r[0]*r[0]
    vector=[]
    for r in res:
         vector.append([r[0]/(norm_sq**0.5)])

print "Maximum eigenvalue (in absolute value):",norm_sq**0.5
print "Eigenvector for the maximum eigenvalue:",vector

#OUTPUT
Maximum eigenvalue (in absolute value): 2.17008648663
Eigenvector for the maximum eigenvalue: [[0.5227207256439814],
[0.6116284573553772], [0.2818451988548684], [0.5227207256439814]]
```

Starting from the data previously downloaded from the laboratory for web algorithmics of the University of Milan, we can now apply the HITS algorithm to the real case of the ".eu" portion of the WWW in 2005. The output will be the top urls and the corresponding auth and hub values.

HITS algorithm for the ".eu" domain in 2005

```
import operator

(auth,hub)=HITS_algorithm(eu_DG_SCC)
sorted_auth = sorted(auth.items(), key=operator.itemgetter(1))
sorted_hub = sorted(hub.items(), key=operator.itemgetter(1))

#top ranking auth
```

```
print "Top 5 auth"
for p in sorted_auth[:5]:
    print dic_nodid_urls[p[0]],p[1]

#top ranking hub
print "\nTop 5 hub"
for p in sorted_hub[:5]:
    print dic_nodid_urls[p[0]],p[1]

#OUTPUT
top 5 auth
http://www.etf.eu.int/WebSite.nsf/... 9.67426387995e-05
http://www.etf.eu.int/website.nsf/Pages/Job... 9.67426387995e-05
http://www.etf.eu.int/WebSite.nsf/(tenders... 9.67426387995e-05
http://europa.eu.int/eures/main.jsp?...LV 9.67426387995e-05
http://europa.eu.int/eures/main.jsp?...DE 9.67426387995e-05

top 5 hub
http://www.etf.eu.int/... 7.65711101121e-07
http://ue.eu.int/cms3_fo/showPage.asp... 7.65711101121e-07
http://ue.eu.int/showPage.asp?id=357... 7.65711101121e-07
http://ue.eu.int/showPage.asp?id=370... 7.65711101121e-07
http://www.europarl.eu.int/interp... 7.65711101121e-07
```

4.3.3 PageRank

HITS is not the only algorithm that assesses the importance of a page by using the spectral properties of the adjacency matrix (or functions of it). Actually, the most successful measure of eigenvector centrality is given by another algorithm, known as PageRank. The idea is similar to that of the HITS algorithm, but now we give only one score to the pages of the web, irrespective of its role as authority or hub. The values of PageRank for the various pages in the graph are given by the eigenvector $\mathbf{r}$, related to the largest eigenvalue λ_1 of the matrix $\mathbf{P}$, given by

$$\mathbf{P} = \alpha\mathbf{N} + (\mathbf{1} - \alpha)\mathbf{E}; \tag{4.6}$$

the weight is taken as $\alpha = 0.85$ in the original paper (Page *et al.*, 1999). $\mathbf{N}$ is the normalised matrix $\mathbf{N} = \mathbf{AK0^{-1}}$ where $\mathbf{A}$ is the adjacency matrix and $\mathbf{K0^{-1}}$ is the diagonal matrix, whose entries on the diagonal are given by the inverse of the out degree, $\left(\mathbf{K0^{-1}}\right)_{ii} = 1/k_i^o$.

This new matrix $\mathbf{P}$ does not differ considerably from the original one $\mathbf{N}$, but has the advantage that (thanks to its irreducibility) its eigenvectors can be computed by a simple iteration procedure Langville and Meyer (2003).

As we have seen for HITS, dealing with a matrix that is not primitive presents a series of problems. The presence of dangling nodes avoids $\mathbf{N}$ being a stochastic matrix

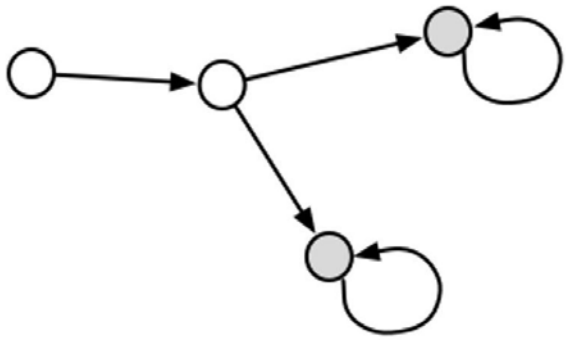


Fig. 4.2 A simple case of reducible matrix.

and therefore gives problems for the existence of the limiting vector $\mathbf{r}^{\infty}$, which is the numerical solution of the equation $\mathbf{r} = \mathbf{rN}$. Even worse, almost certainly, the subgraph represented by $\mathbf{N}$ will be reducible. A reducible stochastic matrix is one for which the underlying chain of transformations is reducible[3]. A reducible chain, is one for which there are states in which the evolution can be trapped. The simplest example of a reducible matrix is that of a page i that has an edge to page j, and this page j has a loop (citing itself) and a link to another page z which again has a loop (an edge to itself) (see Fig. 4.2). Iteration on this set will not produce convergence to a limit vector r^{∞}. When the matrix is *irreducible*, a mathematical theorem (by Perron and Frobenius) ensures that this chain must have a unique and positive stationary vector $\mathbf{r}^{\infty}$ (Perron, 1907; Frobenius, 1912). A physical way to force irreducibility numerically is to destroy the possibility of getting trapped. If you can jump out from a page to a completely random different one (even with small probability), the matrix is irreducible and you can find the eigenvectors $\mathbf{r}$ by iteration. This corresponds to adding to the matrix $\mathbf{N}$ another diagonal matrix $\mathbf{E}$ whose entries e_{ii} are given by $1/n$, where n is the number of vertices in the graph.[4]

Compute the PageRank

```
def pagerank(graph, damping_factor=0.85, max_iterations=100,
min_delta=0.00000001):

    nodes = graph.nodes()
    graph_size = len(nodes)
    if graph_size == 0:
        return {}

    # itialize the page rank dict with 1/N for all nodes
```

[3]In this case the chain is called a Markov chain, since the state at a certain time of evolution depends only upon the state at the previous time step.

[4]In the more recent implementation of PageRank, those entries are actually different from each other, even if they have the same order of magnitude. This is done in order to introduce an ad hoc weight for the different pages.

```
    pagerank = dict.fromkeys(nodes, (1.0-damping_factor)*1.0/ \
                             graph_size)
    min_value=(1.0-damping_factor)/len(nodes)

    for i in range(max_iterations):
        diff = 0 #total difference compared to last iteraction
        # computes each node PageRank based on inbound links
        for node in nodes:
            rank = min_value
            for referring_page in graph.predecessors(node):
                rank += damping_factor * pagerank[referring_page]/ \
                len(graph.neighbors(referring_page))
            diff += abs(pagerank[node] - rank)
            pagerank[node] = rank

        #stop if PageRank has converged
        if diff < min_delta:
            break

    return pagerank
```

Starting with the following test network, we can apply the Pagerank algorithm with both our code and the NetworkX corresponding function.

PageRank for a test network

```
G=nx.DiGraph()
G.add_edges_from([(1,2),(2,3),(3,4),(3,1),(4,2)])
#plot the network
nx.draw(G)

#our Page Rank algorithm
res_pr=pagerank(G,max_iterations=10000,min_delta=0.00000001, \
                damping_factor=0.85)
print res_pr

#Networkx Pagerank function
print nx.pagerank(G,max_iter=10000)

#OUTPUT
{1: 0.17359086186340225, 2: 0.33260446516778386,
3: 0.3202137953926163, 4: 0.17359086304186191}
```

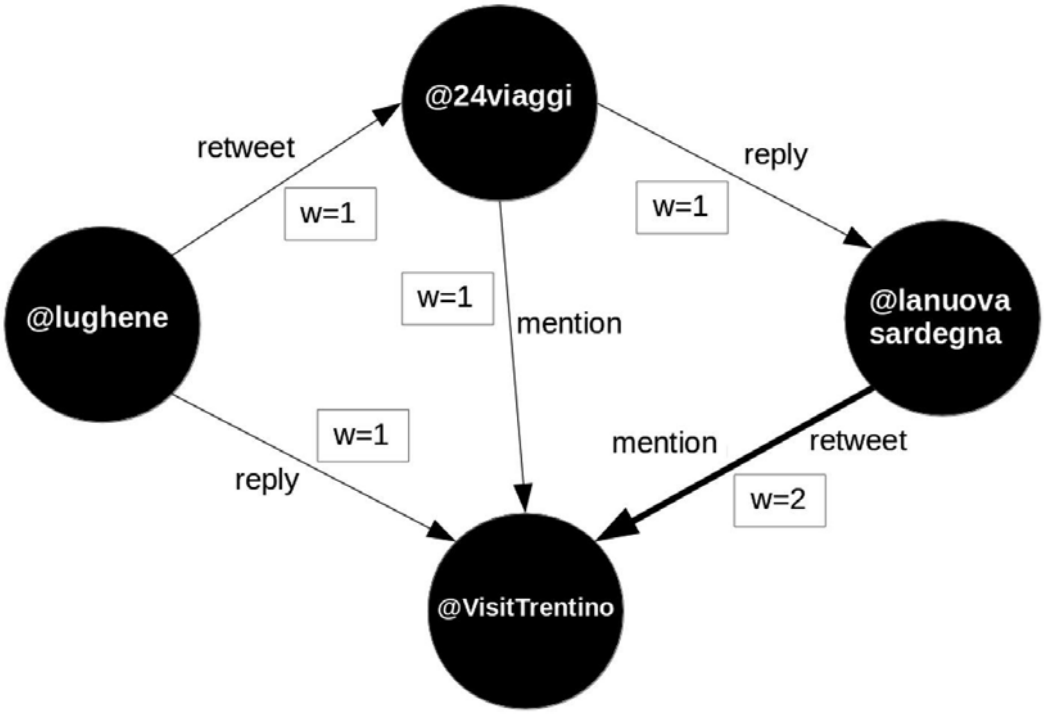


Fig. 4.3 This is the procedure to generate a network starting from a flux of tweets. The nodes are the twitter users and each time one of them mentions, retweets or replies to another user a link is drawn from the first to the second. The weight of a links is the number of citations between the two.

```
{1: 0.17359061775974502, 2: 0.33260554622228633,
3: 0.3202132182582236, 4: 0.17359061775974502}
```

Now we have the opportunity to make sense of this famous algorithm in a real case in the field of social network analysis. At the beginning of this chapter we learnt how to retrieve tweets through the Twitter API. With the "search" method we can get a certain number of tweets and try to uncover the discussion going on related to the search criteria we have imposed. The first step in this process is to map this flux of information in the shape of a network and after that try to measure on it some particular property. More than the so-called structural network of followers and friends, in order to discover the thread of discussions it is useful to generate the network of mentions, retweets, and replies. In this case a link is drawn from a user "A" towards a user "B" if the user "A" mentions user "B" in one of their tweets (see Fig. 4.3).

It is a way of acknowledging someone and giving credit to them, just as happens when a web page relates to another through a hyperlink. This kind of network is able to catch the just-in-time interaction among users about a particular topic, much more than the structural ones. For the present example we will limit ourselves to the mention network only.

Given this procedure we can now apply it to the case of a particular topic and extract in a natural way the thread of the discussion as the clusters that emerge as isolated components of the resulting network (see Fig. 4.4).

Generate and plot the Twitter mention network

```
def generate_network(list_mentions):
    DG=nx.DiGraph()
    for l in list_mentions:
        if len(l)<2: continue
        for n in l[1:]:
             if not DG.has_edge(l[0],n):
                DG.add_edge(l[0],n, weight=1.0 )
        else:
                DG[l[0]][n]['weight']+=1.0
    return DG

#extracting user and mentions for each tweet
res=twitter_connection.search(q='#FutureDecoded', count=5000)
#the first will be the tweer user
list_users={}
list_mentions=[]
for t in res['statuses']:
    list_unique_ids=[]
    print "User Screen Name and Id:",(t[u'user'][u'screen_name'], \
                                       t[u'user'][u'id_str'])
    list_unique_ids.append(t[u'user'][u'id_str'])
    if not list_users.has_key(t[u'user'][u'id_str']):
        list_users[t[u'user'][u'id_str']]=t[u'user'][u'screen_name']
    print "List of Mentions:",
    for m in t[u'entities'][u'user_mentions']:
        if m['id_str']!=t[u'user'][u'id_str']:
                list_unique_ids.append(m['id_str'])
                if not list_users.has_key(m['id_str']):
                    list_users[m['id_str']]=m[u'screen_name']
        print (m[u'screen_name'],m['id_str']),
    print "\r"
    print list_unique_ids
    list_mentions.append(list_unique_ids)
    print "\n"

net_mentions=generate_network(list_mentions)

#plotting the network
```

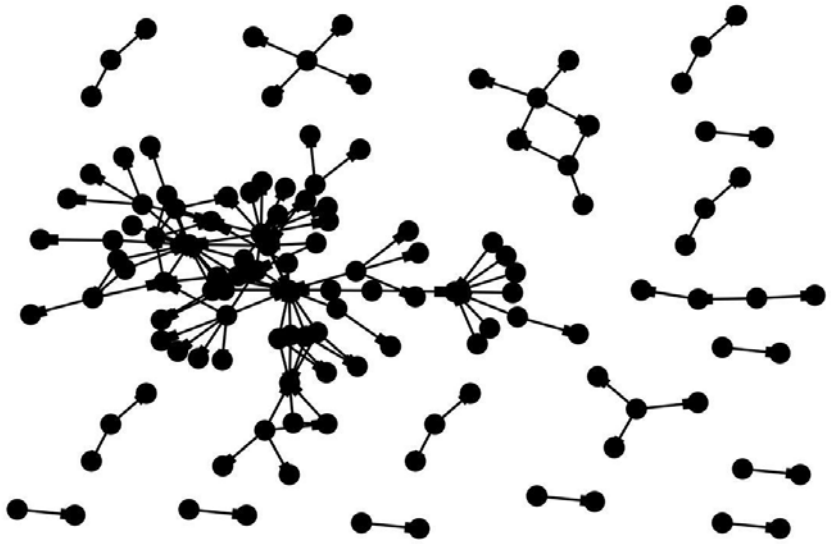


Fig. 4.4 Plot of the mention network arising from Twitter data. The nodes are the Twitter users and the oriented links go from one user to another, mentioned by the first in one of their tweets. In a natural way the threads emerge from the picture as isolated clusters, with a main one dominating the discussion.

```
pos=nx.graphviz_layout(net_mentions,prog='neato')
nx.draw(net_mentions, pos, node_size = 50, node_color='Black')
savefig('./data/hashtag_discussion_thread.png',dpi=600)

#OUTPUT
...
User Screen Name and Id: (u'vincent_salmon', u'294176299')
List of Mentions: (u'TagetikUK', u'3653601856') (u'manuelvellutini',
u'139816639') (u'MicrosoftUK', u'720474368') (u'mspartnersuk',
u'23672986') [u'294176299', u'3653601856', u'139816639',u'720474368',
u'23672986']
...
```

Finally, we can compute the Pagerank on this oriented network getting the most central nodes (users). In this case the simple interpretation is that the top ranking users are very likely the most influential(see also (Perra *et al.*, 2009)), in relation to the selected topic.

Top Pageranks on a Twitter generated network (influencers)

```
pr=nx.pagerank(net_mentions,max_iter=10000)
sorted_pr=sorted(pr.items(), key=operator.itemgetter(1),reverse=True)
#top10 pagerank twitter user from the selected search
for page in sorted_pr[:10]:
    print list_users[page[0]],page[1]

#OUTPUT
microsoftitalia 0.0504950565972
GiacomoFrisoni 0.0480209549756
satyanadella 0.0359068749876
MSFTBusinessUK 0.0350727368583
FabioSantini71 0.032280683387
Microsoft 0.0234986623942
federicadestr 0.022313707312
purassan 0.0177527343108
msdev_ita 0.0109019463377
Fagrossi67 0.0106376988422
```

4.4 Communities and Girvan–Newman algorithm

The concept of communities is not in itself extremely precise, and also therefore methods for determining them in networks are many and refer to slightly different objects. Actually, we can have communities of people corresponding to connected subgraphs of the graph (similar to cliques). On the other hand we can define communities by means of vertices with similar properties (i.e. sharing similar links). In this latter case a community can be determined by a set of vertices which may be totally disconnected. Loosely speaking, when dealing with a large graph we would be interested in a series of vertices and edges that are all somewhat "similar". That is to say we would like to be able to determine some "thematic" subgraphs out of the original (larger) one. Unfortunately, there are various ways of obtaining such a partition of the graph, and *a priori* we cannot ensure that one is better than the others, so it is impossible to tell which method must be used to determine network communities. Imagine the situation of a bipartite graph (i.e. authors on one side and papers on the others). By construction, in such a structure the authors do not have any shared edge, and a community is rather defined by the papers that they are connected to. On the other hand we can transform this graph to an author–author graph where the links connect the persons writing a paper together. In this case the same community is determined by considering which vertices are more closely connected with each other, so that the common edges play a crucial role.

In the following we shall mostly stick to the latest definition of community; showing methods and techniques to determine which subgraphs are composed of vertices which

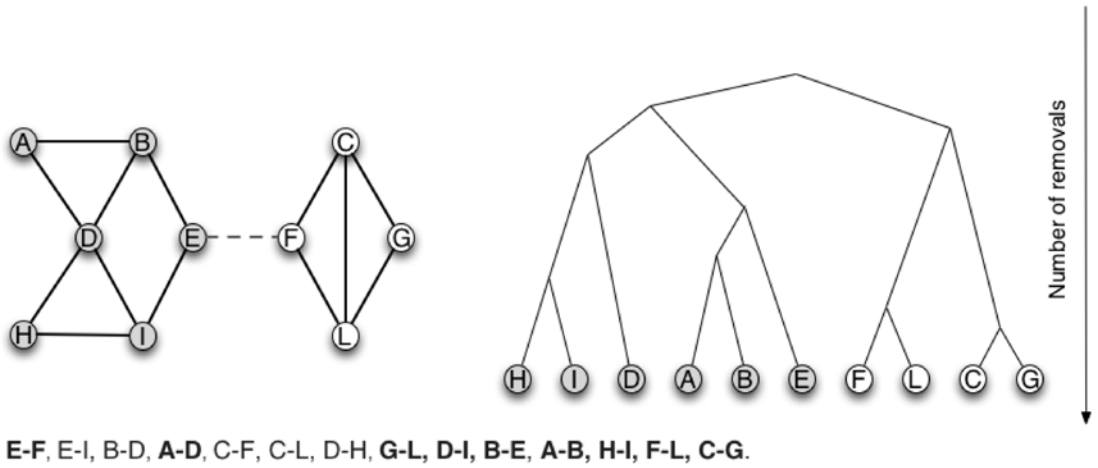


Fig. 4.5 (left) A toy graph to which we applied the GN algorithm. First we compute the edge betweenness and then we cut the edge with the largest value (dashed). Recursively, we compute and again delete all the edges one after another. Whenever the removal of one edge splits the graph, we indicate (right) the edge in bold (i.e. edges **E-F, A-D, G-L, D-I, B-E, A-B, H-I, F-L, C-G**). As a result we obtain the dendrogram on the right.

are closely connected with each other with a strength or density of links larger than the average.

If this is the case in a connected graph, it turns out that communities are denser subgraphs connected by a few links that act as a bridge between them. By cutting out these bridges communities emerge as isolated subgraphs. This is the main idea behind the divisive method of Girvan and Newman (Girvan and Newman, 2002).

4.4.1 Girvan–Newman (GN) algorithm

This method of computing communities, is based on a recursive deletion of edges (Girvan and Newman, 2002; Newman and Girvan, 2004). These edges are not randomly chosen, rather, they are selected for their bridging properties, that is to say they are selected if they connect dense regions and therefore after their removal these dense regions appear as the communities within the system. The quantity chosen for this procedure is the edge betweenness (see Section 3.3). Based on this measure of centrality, one computes the betweenness on all the edges of the graph. We start removing the edge with the largest value then we recompute the edge betweenness and then we delete the one with the largest betweenness among those left. The process is repeated until all the edges are removed. Somewhere during this procedure the structure of communities emerges, however, at different stages we have different sets of communities that vary in both the number of clusters and their size. An example of the procedure is sketched out in Fig. 4.5.

We use a simple toy graph to work out the procedure. We compute the edge betweenness of all the links in the graph. Then we start removing the largest one. Whenever the graph splits into two parts, we keep track of it in the dendrogram. Often, especially at the end of the process, many edges have the same (largest) value

of betweeness; in this case we select on of them randomly. This recursive procedure finishes when all the vertices are disconnected. The main problem associated with this way of computing communities, is knowing when to stop the process before we split the network into isolated vertices. Various implementations have been made of this method. For example for large graphs one can compute the betweenness, not by considering all the couples vertices, but just a random selection of the vertices (Tyler *et al.*, 2003). This results in an effective gain in the speed of the algorithm, paying the price of reduced precision.

Code for the GN algorithm

```
G=nx.Graph()
G.add_edges_from([('A','B'),('A','D'),('B','D'),('B','E'),('E','I'),\
                  ('D','I'),('D','H'),('H','I'),('E','F'),('F','C'),\
                  ('F','L'),('C','L'),('C','G'),('G','L')])

pos=nx.graphviz_layout(G,prog='neato')

nx.draw(G, pos,with_labels=True)

#NOTE: THE ORDER OF EDGES IS DIFFERENT FOR THE FACT THAT MANY
#OF THEM HAVE THE SAME BETWEENNESS VALUE...

sorted_bc=[1]
actual_number_components=1
while not sorted_bc==[]:
    d_edge=nx.edge_betweenness_centrality(G)
    sorted_bc = sorted(d_edge.items(), key=operator.itemgetter(1))
    e=sorted_bc.pop()
    print "deleting edge:", e[0],
    G.remove_edge(*e[0])
    num_comp=nx.number_connected_components(G)
    print "...we have now ",num_comp," components"
    if num_comp>actual_number_components:
        actual_number_components=num_comp

#OUTPUT
deleting edge: ('E', 'F') ...we have now  2  components
deleting edge: ('B', 'E') ...we have now  2  components
deleting edge: ('D', 'I') ...we have now  2  components
deleting edge: ('D', 'H') ...we have now  3  components
deleting edge: ('I', 'H') ...we have now  4  components
deleting edge: ('F', 'L') ...we have now  4  components
deleting edge: ('C', 'F') ...we have now  5  components
```

```
deleting edge: ('B', 'D') ...we have now  5  components
deleting edge: ('A', 'B') ...we have now  6  components
deleting edge: ('G', 'L') ...we have now  6  components
deleting edge: ('C', 'G') ...we have now  7  components
deleting edge: ('A', 'D') ...we have now  8  components
deleting edge: ('C', 'L') ...we have now  9  components
deleting edge: ('E', 'I') ...we have now  10  components
```

4.5 Modularity

The whole idea behind the GN algorithm is that the communities are the set of subgraphs that have a link density larger than "expected" (for a random graph of the same size and measure). By cutting bridging edges we isolate such communities and we are able to determine them quantitatively. Since this process does not tell us when one division is better than another, we need a quantity for assessing how good the division is and therefore when we should stop. This quantity is called *modularity* (Newman, 2006) and it assigns a score to any division in clusters one obtains from a given graph. The steps we need to take in order to define this quantity are as follows:

- the starting point is to consider a partition of the graph into g subgraphs;
- if the partition is good most of the edges will be inside the subgraphs and few will connect them;
- we then define a $g \times g$ matrix E whose entries e_{ij} give the fraction of edges that in the original graph connect subgraph i to subgraph j;
- the actual fraction of edges in subgraph i is given by element e_{ii};
- the quantity $f_i = \sum_{j=1,g} e_{ij}$ gives the probability that an end-vertex of a randomly extracted edge is in subgraph i ($i \in 1, ..., g$);
- in the absence of correlations the probability that an edge belongs to subgraph i is f_i^2.

We can now define the modularity Q of a given partition by considering the actual distribution of edges in the partition, with respect to the one we have for a random case, i.e.

$$Q = \sum_{i=1}^{g} e_{ii} - f_i^2, \tag{4.7}$$

which represents a measure of the validity of a certain partition of the graph. In the limit case where we have a random series of communities, the edges can be with the same probability in the same subgraph i or between two different subgraphs i, j. In this case $e_{ii} = f_i^2$ and $Q = 0$. If the division into subgraphs is appropriate, then the actual fraction of internal edges e_{ii} is larger than the estimate f_i^2, and the modularity is larger than zero. Surprisingly, random graphs (which, as we shall see, are graphs obtained by randomly drawing edges between vertices) can present partitions with large modularity (Guimerà *et al.*, 2004). In random networks of finite size it is possible

to find a partition which not only has a nonzero value of modularity, but even quite high values. For example, a network of 128 nodes and 1024 edges has a maximum modularity of 0.208. While on average we expect a null modularity for a random graph, this does not exclude that by careful choice we can obtain a different result. This suggests that those networks that seem to have no structure actually exhibit community structure due to fluctuations.

Community detection with the Karate Club network (See Fig. 4.6)

```
import community

G=nx.read_edgelist("./data/karate.dat")

#first compute the best partition
partition = community.best_partition(G)

#plot the network
size = float(len(set(partition.values())))
pos = nx.spring_layout(G)
count = 0.
plt.axis('off')
for com in set(partition.values()) :
    count = count + 1.
    list_nodes = [nodes for nodes in partition.keys() \
                  if partition[nodes] == com]
    nx.draw_networkx_nodes(G, pos, list_nodes, node_size = 300, \
                           node_color = str(count / size))
    nx.draw_networkx_labels(G,pos)

nx.draw_networkx_edges(G,pos, alpha=0.5,width=1)
savefig('./data/karate_community.png',dpi=600)
```

We can perform the same analysis on the Sardinian Wikipedia with the aim of extracting the relevant communities. The first thing to do is to load the network and define the dictionary that associates the Wikipedia node_ids with the page titles.

Community detection for the scwiki web graph

```
#load the directed and undirected version og the scwiki graph
scwiki_pagelinks_net_dir=nx.read_edgelist \
("./data/scwiki_edgelist.dat",create_using=nx.DiGraph())
scwiki_pagelinks_net=nx.read_edgelist("./data/scwiki_edgelist.dat")
```

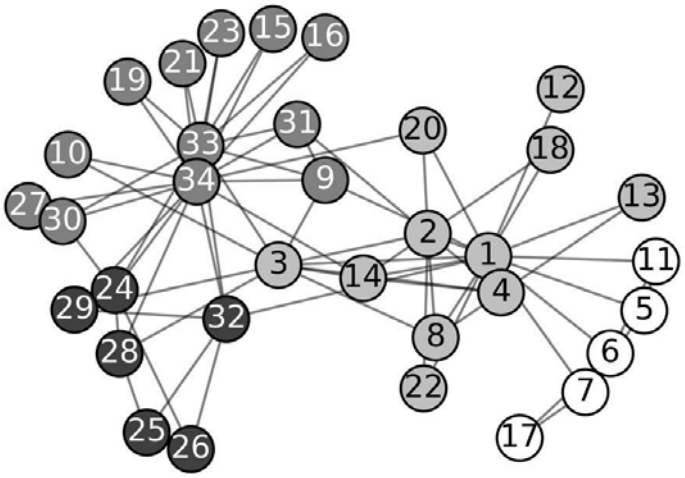


Fig. 4.6 The Karate Club social network after optimisation of the modularity function.

```
#load the page titles
diz_titles={}
file_titles=open("./data/scwiki_page_titles.dat",'r')
while True:
    next_line=file_titles.readline()
    if not next_line:
        break
    print next_line.split()[0],next_line.split()[1]
    diz_titles[next_line.split()[0]]=next_line.split()[1]

file_titles.close()

#OUTPUT
...
4311 Lod
7157 Logos_Bascios
13786 Logroo
8490 Logudoresu
4548 Logudoro
4825 Logusantu
4900 Loiri-Poltu_Santu_Paolu
...
```

The problem in plotting this network is that it comprises almost 10,000 nodes. To overcome this problem we generate a representative network in which each node is a community (we consider just the first nine with more than 200 nodes), with size proportional to the number of nodes in the corresponding community and edge weight proportional to the number of edges between each pair of communities (we cut the link below the threshold weight 100). The representative node is chosen according to the Pagerank inside the corresponding community. So as a first step we generate the representative nodes and print the association with the page title of the scwiki Wikipedia page and the edges with the appropriate weights. The output will be the community id, the number of nodes in it, the page title, and the corresponding PageRank.

Generate and optimise the representative network of the community structure

```
#optimization
partition = community.best_partition(scwiki_pagelinks_net)

#Generate representative nodes of the community structure
community_structure=nx.Graph()
diz_communities={}
diz_node_labels={}
diz_node_sizes={}
max_node_size=0
for com in set(partition.values()) :
    diz_communities[com] = [nodes for nodes in partition.keys() \
                            if partition[nodes] == com]
    if len(diz_communities[com])>=200:
        if max_node_size<len(diz_communities[com]):
            max_node_size=len(diz_communities[com])
        print "community",com,len(diz_communities[com]),
        sub_scwiki_dir = scwiki_pagelinks_net_dir.subgraph \
        (diz_communities[com])
        res_pr=nx.pagerank(sub_scwiki_dir,max_iter=10000)
        sorted_pr=sorted(res_pr.items(), key=operator.itemgetter \
                         (1),reverse=True)
        print diz_titles[sorted_pr[0][0]],sorted_pr[0][1]
        community_structure.add_node(com)
        diz_node_labels[com]=diz_titles[sorted_pr[0][0]]
        diz_node_sizes[com]=len(diz_communities[com])

#Generate edge weights according to the number of links
#among communities
max_edge_weight=0.0
for i1 in range(community_structure.number_of_nodes()-1):
```

```
    for i2 in range(i1+1,community_structure.number_of_nodes()):
        wweight=0.0
        for n1 in diz_communities[community_structure.nodes()[i1]]:
            for n2 in diz_communities[community_structure.nodes() \
                                      [i2]]:
                if scwiki_pagelinks_net.has_edge(n1,n2):
                    wweight=wweight+1.0
        if wweight>100.0:
            if max_edge_weight<wweight:
                max_edge_weight=wweight
            community_structure.add_edge(community_structure. \
            nodes()[i1],community_structure.nodes()[i2], \
                                        weight=wweight)

#OUTPUT

community 0 2012 Logudoresu 0.0507648029074
community 1 1812 Wikipedia 0.0498390890911
community 2 861 Classificatzione_sientfica 0.0358451735252
community 3 795 Babel 0.0296037336974
community 4 662 Sardigna 0.0403623149927
community 5 393 Nugoresu 0.0781774191072
community 6 1939 Limba_Sarda_Comuna 0.0219626233778
community 8 201 Rabascius 0.053309971387
community 22 223 Casteddu 0.0197102893607
```

The final plot will reveal the hierarchy of the community structure, the relative sizes of nodes/communities and edges (see Fig. 4.7). This result shows that the most important nodes in the communities are related to territorial locations and language classifications, topics that are known to be relevant to the Sardinian culture.

Plotting the representative network of the community structure

```
pos=nx.graphviz_layout(community_structure,prog='circo')
node_size_factor=2000.0
edge_weight_factor=10.0

plt.axis('off')

for n in community_structure.nodes():
    nx.draw_networkx_nodes(community_structure, pos, [n], node_size\
                           = node_size_factor*diz_node_sizes[n]/ \
                           max_node_size, node_color='Black')
```

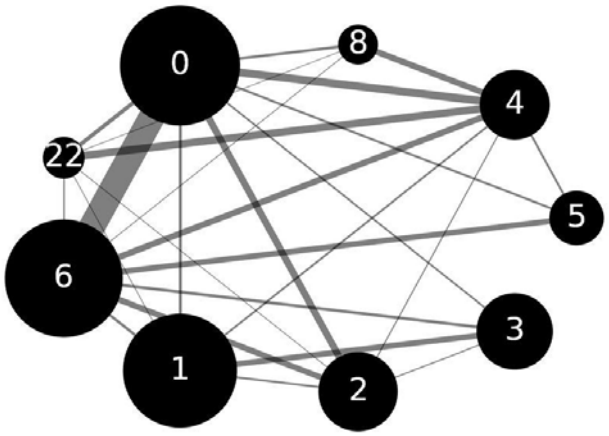


Fig. 4.7 Network representation of the community structure of the Sardinian Wikipedia scwiki. We selected the main communities (the first nine, with more than 200 nodes) the size of each node being proportional to the number of nodes in each community. The edge width is instead proportional to the number of edges between each pair of communities.

```
    nx.draw_networkx_labels(community_structure,pos, font_color= \
                            'White',axis='off')

for e in community_structure.edges():
    nx.draw_networkx_edges(community_structure,pos,[e],alpha=0.5, \
                           width=edge_weight_factor* \
                           community_structure[e[0]][e[1]]['weight']\
                           /max_edge_weight)
```