In this session:

- You are going to learn about the igraph package
- You are going to compute several descriptive measures of networks and nodes
- You are going to run community detection algorithms and explore their results

1 Introduction

In this session we will introduce the igraph software package for network analysis. This guide uses R as the platform to using igraph, but there exist python bindings for igraph as well in case you prefer to use that. RStudio is an excellent IDE for R.

There are igraph bindings for Python and C++; it should be easy to translate the code in this guide into these other languages.

R is "a language and environment for statistical computing and graphics"[1] with a very active community of contributors. Available from http://www.r-project.org/.

RStudio is "a free and open source integrated development environment for R"[2]. It makes working with R more pleasant. Available from http://www.rstudio.com/.

igraph is "a free software package for creating and manipulating undirected and directed graphs"[3]. Available from http://igraph.sourceforge.net/.

The computers in the PC Lab should have these three components installed. If igraph is not installed, then you can do so through RStudio's install manager or directly through the command line with the install.packages instruction. Then, to load the library so that you can use its functionality you should introduce the following command into the console:

```
> library(igraph)
```

[1] From http://www.r-project.org/about.html
[2] From http://www.rstudio.com/ide/
[3] From http://igraph.sourceforge.net/introduction.html

2 Basics

This section will cover the basic commands for creating, manipulating and visualizing graphs using `igraph`. It should also help as an introduction to the main `R` commands. If you are unfamiliar with `R`, there are many online tutorials. However, for this session there is very little programming involved so you are not required to learn a new programming language from scratch.

2.1 Creating graphs

The objects we study in this course are *graphs* (or *networks*). They consist of a set of *nodes* and a set of *edges*. As an example, if you type into the `RStudio` console the following command

```
g <- graph( c(1,2, 1,3, 2,3, 3,5), n=5 )
```

In this command, we are assigning to the variable g a graph that has nodes $V = \{1,2,3,4,5\}$ and has edges $E = \{(1,2),(1,3),(2,3),(3,5)\}$

The commands `V(g)` and `E(g)` print the list of nodes and edges of the graph g:

```
> V(g)
Vertex sequence:
[1] 1 2 3 4 5
> E(g)
Edge sequence:

[1] 1 -> 2
[2] 1 -> 3
[3] 2 -> 3
[4] 3 -> 5
```

You can add nodes and edges to an already existing graph, e.g.:

```
> g <- graph.empty() + vertices(letters[1:10], color="red")
> g <- g + vertices(letters[11:20], color="blue")
> g <- g + edges(sample(V(g), 30, replace=TRUE), color="green")
> V(g)
Vertex sequence:
 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p" "q" "r" "s" "t"
> E(g)
Edge sequence:

[1]  q -> q
```

```
[2]  n -> a
[3]  j -> p
[4]  s -> h
[5]  f -> c
[6]  h -> f
[7]  g -> r
[8]  t -> t
[9]  j -> o
[10] c -> f
[11] i -> a
[12] o -> q
[13] c -> j
[14] r -> i
[15] a -> b
```

These lines create a graph g with 20 nodes and 15 random edges. Notice that nodes and edges can also have attributes, e.g. in this example we are assigning different colors to nodes. The command `sample` returns a vector containing a sample of 30 random vertices from g.

2.1.1 Loading graphs

We have seen how to create graphs from scratch, but most often we will be loading them from a file containing the graph in some sort of format. `igraph` handles many graph formats already. The simplest one is a file containing the edge list. For example, suppose that we have a file `list.txt` containing the following three edges:

```
0 1
1 2
2 3
```

We can create a graph using the command

```
> g <- read.graph("graph.txt", format="edgelist")
> V(g)
Vertex sequence:
[1] 1 2 3 4
> E(g)
Edge sequence:

[1] 1 -> 2
[2] 2 -> 3
[3] 3 -> 4
```

Notice that the node ids within igraph start with 1, but the input file expects the first id to be 0. We believe this is a bug in the implementation of igraph, but you should keep this in mind.

We can also access online graphs, e.g. the following command loads a *Pajek* graph from an online site

```
karate <- read.graph("http://cneurocvs.rmki.kfki.hu/igraph/karate.net", format="pajek")
```

2.1.2 Graph generators

igraph implements also many useful graph generators. We have already seen a few models in class, in particular: the Edös-Rényi model (ER), the Barabasi-Albert model (BA), and the Watts-Strogratz model (WS). The following commands generate graphs using these models:

```
er_graph <- erdos.renyi.game(100, 2/100)
ws_graph <- watts.strogatz.game(1, 100, 4, 0.05)
ba_graph <- barabasi.game(100)
```

2.2 Manipulating attributes in graphs

We can add attributes to nodes and edges of the graphs. These are useful for selecting certain types of nodes, and for visualization purposes.

```
> g <- erdos.renyi.game(10, 0.5)
> V(g)$color <- sample( c("red", "black"), vcount(g), rep=TRUE)
> E(g)$color <- "grey"
> red <- V(g)[ color == "red" ]
> bl <- V(g)[ color == "black" ]
> E(g)[ red %--% red ]$color <- "red"
> E(g)[ bl %--% bl ]$color <- "black"
```

What these commands do is to generate a random graph with 10 nodes, assigns random colors to the nodes, colors edges joining red nodes in red, and edges joining black nodes in black. All remaining edges are colored grey.

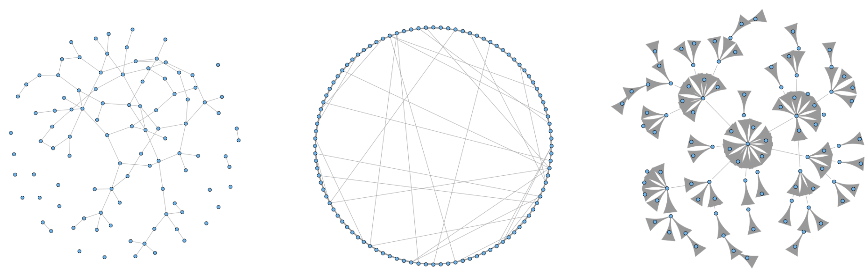
The next example assigns random weights to a lattice graph and then colors the ones having weight over 0.9 red, and the rest grey.

```
> g <- graph.lattice( c(10,10) )
> E(g)$weight <- runif(ecount(g))
> E(g)$color <- "grey"
> E(g)[ weight > 0.9 ]$color <- "red"
```

2.3 Visualizing graphs

A very important part in the analysis of networks is being able to *visualize* them. As an example the following commands render the three graphs depicted in the figure below.

```
> er_graph <- erdos.renyi.game(100, 2/100)
> plot(er_graph, vertex.label=NA, vertex.size=3)
> ws_graph <- watts.strogatz.game(1, 100, 4, 0.05)
> plot(ws_graph, layout=layout.circle, vertex.label=NA, vertex.size=3)
> ba_graph <- barabasi.game(100)
> plot(ba_graph, vertex.label=NA, vertex.size=3)
```



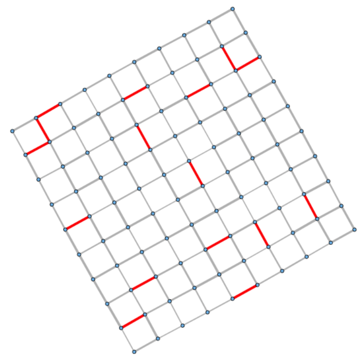
Erdös-Rényi   Watts-Strogatz   Barabási-Albert

The plot command is very flexible and has many parameters that control the behavior of the visualization. You can already see a few in the example above. For example, `vertex.label` controls the label written in the nodes, if set to `NA` then no text label is written. You can access all the parameters and their possible values through the help system by typing

```
> help(igraph.plotting)
```

As another example, consider adding attributes to edges for a nicer visualization:

```
> g <- graph.lattice( c(10,10) )
> E(g)$weight <- runif(ecount(g))
> E(g)$color <- "grey"
> E(g)[ weight > 0.9 ]$color <- "red"
> plot(g, vertex.size=2, vertex.label=NA, layout=layout.kamada.kawai,
edge.width=2+3*E(g)$weight)
```



2.4 Measuring graphs

There are many measures that help us understand and characterize networks. We have seen three in class already: diameter (and average path length), clustering coefficient (or transitivity), and degree distribution. `igraph` provides functions that compute these measures for you. The functions are: `diameter`, `transitivity`, `average.path.length`, `degree`, and `degree.distribution`. The examples below illustrate the usage of these functions.

For `diameter` and `average.path.length`

```
> g <- graph.lattice( length=100, dim=1, nei=4 )
> average.path.length(g)
[1] 8.79798
> diameter(g)
[1] 25
> g <- rewire.edges( g, prob=0.05 )
> average.path.length(g)
[1] 3.132323
> diameter(g)
[1] 6
```

For `transitivity`

```
> ws <- watts.strogatz.game(1, 100, 4, 0.05)
> transitivity(ws)
[1] 0.5466147
> p_hat <- ecount(ws)/(vcount(ws)*(vcount(ws))/2)
> p_hat
[1] 0.08
> er <- erdos.renyi.game(100, p_hat)
> transitivity(er)
```

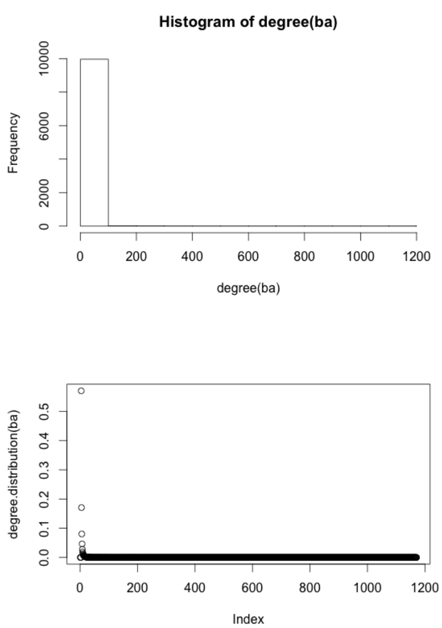
```
[1] 0.08411215
```

For `degree` and `degree.distribution`

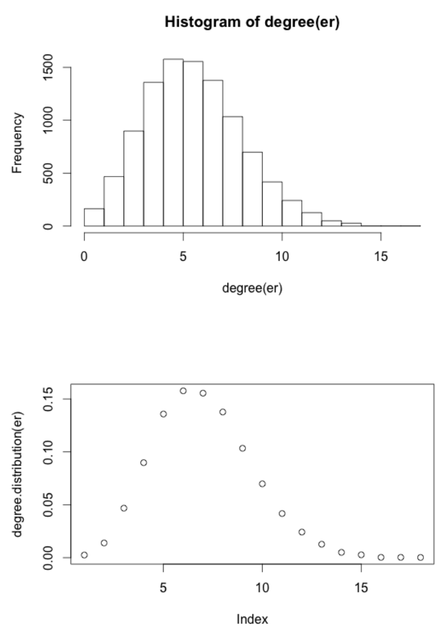
```
> g <- graph.ring(10)
> plot(g)
> degree(g)
 [1] 2 2 2 2 2 2 2 2 2 2

> ba <- barabasi.game(10000, m=3)
> p_hat <- ecount(ba)/ ((vcount(ba)-1)*vcount(ba)/2)
> er <- erdos.renyi.game(10000, p_hat)
> degree.distribution(er)
 [1] 0.0025 0.0139 0.0468 0.0898 0.1358 0.1577 0.1555 0.1377 0.1034 0.0698 0.0417 0.0242
[13] 0.0127 0.0050 0.0027 0.0003 0.0003 0.0002

> hist(degree(er))
> hist(degree(ba))
> plot(degree.distribution(er))
> plot(degree.distribution(ba))
```



Barabási-Albert



Erdös-Rényi

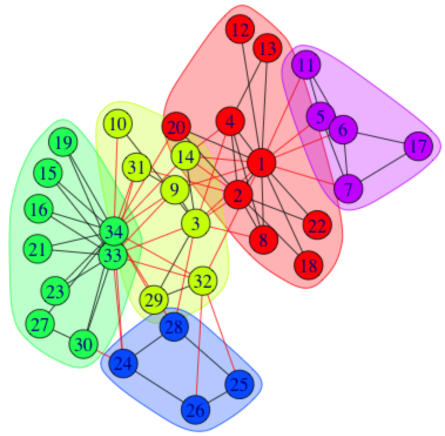
3 Community detection with igraph

In this session you will run and compare different community finding algorithms. In the igraph package there are a few already implemented, including some we have seen in theory class:

- edge.betweenness.community [Newman and Girvan, 2004]
- fastgreedy.community [Clauset et al., 2004] (modularity optimization method)
- label.propagation.community [Raghavan et al., 2007]
- leading.eigenvector.community [Newman, 2006]
- multilevel.community [Blondel et al., 2008] (the Louvain method)
- optimal.community [Brandes et al., 2008]
- spinglass.community [Reichardt and Bornholdt, 2006]
- walktrap.community [Pons and Latapy, 2005]
- infomap.community [Rosvall and Bergstrom, 2008]

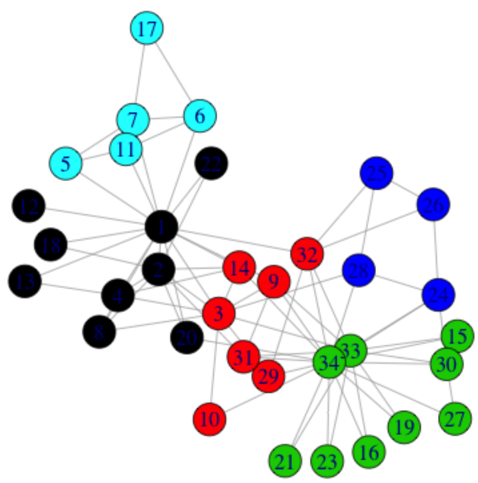
All of these methods return a commmunities object, which you can then use to explore, plot, and compute metrics on. As an example, consider the following snippet of code:

```
> karate <- graph.famous("Zachary")
> wc <- walktrap.community(karate)
> modularity(wc)
[1] 0.3532216
> membership(wc)
 [1] 1 1 2 1 5 5 5 1 2 2 5 1 1 2 3 3 5 1 3 1 3 1 3 4 4 4 3 4 2 3 2 2 3 3
> plot(wc, karate)
```



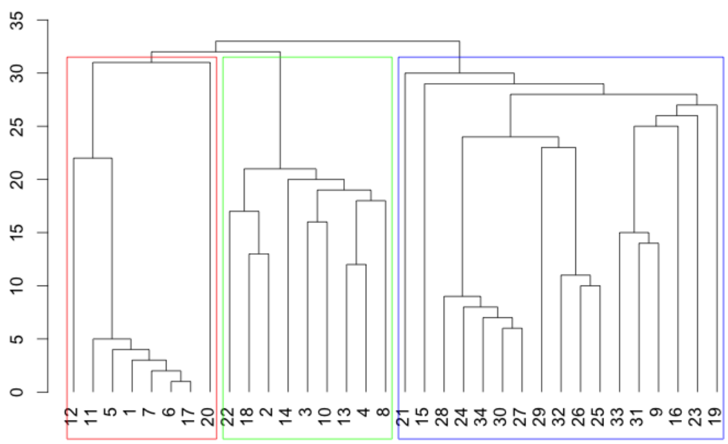
An alternative way of plotting communities without the shaded regions is:

```
> plot(karate, vertex.color=membership(wc))
```



For those algorithms that output communities with hierarchical structure, this information can be visualized using the `dendPlot` function, which displays the corresponding dendogram:

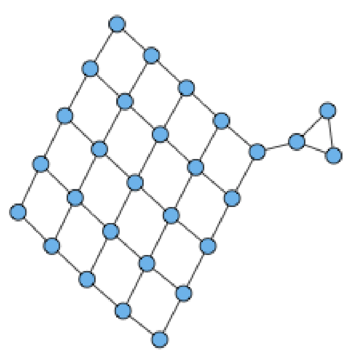
```
> karate <- graph.famous("Zachary")
> fc <- fastgreedy.community(karate)
> dendPlot(fc)
```



4 Your tasks

Before proceeding, make sure you understand the code and examples provided in this guide. Be prepared to consult `igraph`'s help and documentation[4] to carry out the following tasks.

Task 1. Write a script that creates the following graph



[4]http://igraph.sourceforge.net/documentation.html

Task 2. Compute the degree distribution, diameter, average path length, and transitivity of Zachary's famous karate club network.

Task 3. Plot Zachary's famous karate club network and make sure that node sizes are proportional to their page rank values. What vertex has the highest pagerank? Repeat the same but for closeness and betweenness centrality values.

Task 4. Load the network from `wikipedia.gml`[5] provided with this session's files. It is in `gml` format, which can be imported into `igraph` using the command `read.graph("wikipedia.gml", format="gml")`. The vertices of this network are wikipedia pages. The label of each vertex is the title of the wikipedia page. Now, use a community detection algorithm of your choice from the list provided.

1. What is the community containing the vertex with highest pagerank. Print a few of the titles of the pages in this community. What do you think is the topic of the community? Repeat the same but with the vertex with lowest pagerank.
2. How many nodes does the largest community found contain? Plot the histogram of community sizes.
3. Use another community detection algorithm and check whether the community of the vertex with highest and lowest pagerank changes significantly. Based on this, can you say which algorithm worked the best? Contrast your finding with the modularity value obtained by both community outputs.

5 Deliverables

Important rule: The lab session, and especially the report you have to hand in, are strictly individual work. Plagiarism will be prosecuted. Nevertheless, you are encouraged to ask the teacher as soon as possible if you think you don't understand what you are supposed to do, and also if you feel you are spending much more time than the rest of the group – sometimes a tiny error can be tricky to find and doesn't add much to your knowledge. Questions can be asked either in person or by email, and you'll never be penalized by asking questions, no matter how stupid they look in retrospect.

To deliver: You must deliver a brief report describing your results. The formats accepted for the report are, in principle, pdf, Word, OpenOffice, and Postscript. You also have to hand in the script or scripts you used to solve the tasks.

Procedure: Submit your work through the raco platform as a single zipped file.

[5] Thanks to Lada Adamic for providing this in her course *Social Network Analysis*.

Deadline: Work must be delivered within 2 weeks from the lab session you attend. Late submissions risk being penalized or not accepted at all. If you anticipate problems with the deadline, tell me as soon as possible.

References

[Blondel et al., 2008] Blondel, V. D., Guillaume, J.-l., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of community hierarchies in large networks. *Networks*, pages 1–6.

[Brandes et al., 2008] Brandes, U., Delling, D., Gaertler, M., Gorke, R., Hoefer, M., Nikoloski, Z., and Wagner, D. (2008). On Modularity Clustering. *IEEE Transactions on Knowledge and Data Engineering*, 20.

[Clauset et al., 2004] Clauset, A., Newman, M. E. J., and Moore, C. (2004). Finding community structure in very large networks. *Physical Review E*.

[Newman, 2006] Newman, M. E. J. (2006). Finding community structure in networks using the eigenvectors of matrices. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 74:036104.

[Newman and Girvan, 2004] Newman, M. E. J. and Girvan, M. (2004). Finding and evaluating community structure in networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 69(2 Pt 2):026113.

[Pons and Latapy, 2005] Pons, P. and Latapy, M. (2005). Computing communities in large networks using random walks. *Journal of Graph Algorithms and Applications*, 10:191–218.

[Raghavan et al., 2007] Raghavan, U. N., Albert, R., and Kumara, S. (2007). Near linear time algorithm to detect community structures in large-scale networks. *Physical review. E, Statistical, nonlinear, and soft matter physics*, 76:036106.

[Reichardt and Bornholdt, 2006] Reichardt, J. and Bornholdt, S. (2006). Statistical mechanics of community detection. *Physical Review E*, 74.

[Rosvall and Bergstrom, 2008] Rosvall, M. and Bergstrom, C. T. (2008). Maps of random walks on complex networks reveal community structure. *Proceedings of the National Academy of Sciences of the United States of America*, 105:1118–1123.