Introduction to `igraph` (v. 1.2.5)

Argimiro Arratia & Ramón Ferrer

September 21, 2020

1 Introduction

This session will introduce the basic tools for the analysis of networks that we will be using during this course. We will use the following software:

`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 `https://cran.r-project.org/web/packages/igraph/`. Documentation (a manual of 458 pages!): `https://cran.r-project.org/web/packages/igraph/igraph.pdf`

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. Then, to load the library so that you can use its functionality you should introduce the following command into the console:

```
> library(igraph)
```

You are expected to be able to consult help online so that you can achieve the tasks you are required to do, even if you have never used this software before.

[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; you can find many using Google, for example.

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 `graph.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. This is because `igraph` enforces consecutive nume-

rical ids for vertices (and edges) and always starting at 1. If some operation changes the number of vertices in the graphs, e.g. a subgraph is created via `induced_subgraph`, then the vertices are renumbered to satisfy this criteria. Keep this in mind!

For example, run and see the results of the following commands:

```
> IG <- induced.subgraph(g,2:4)
> V(IG)
```

If you really want the node id to be its *label* (or to work with specific labels for the vertices), you can set this with `set_vertex_attribute`:

```
> g=set_vertex_attr(g,"name", value = 0:3)
```

Now, list again the vertices and edges and you will see the change.

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.

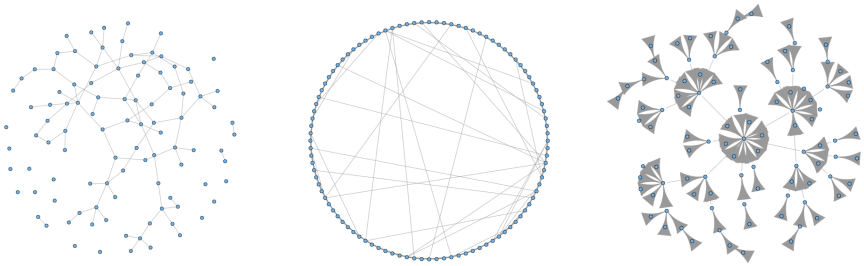
The next example assigns random weights to a lattice graph and then colors the ones having weight over 0.9 red, and the rest grey.

```
> 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)
```



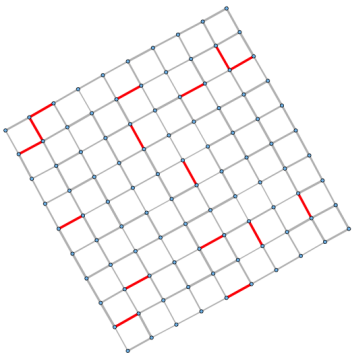
Erdös-Rényi Watts-Strogatz Barabási-Albert

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( g, each_edge( prob=0.05 ))
> average.path.length(g)
[1] 3.132323
> diameter(g)
[1] 6
```

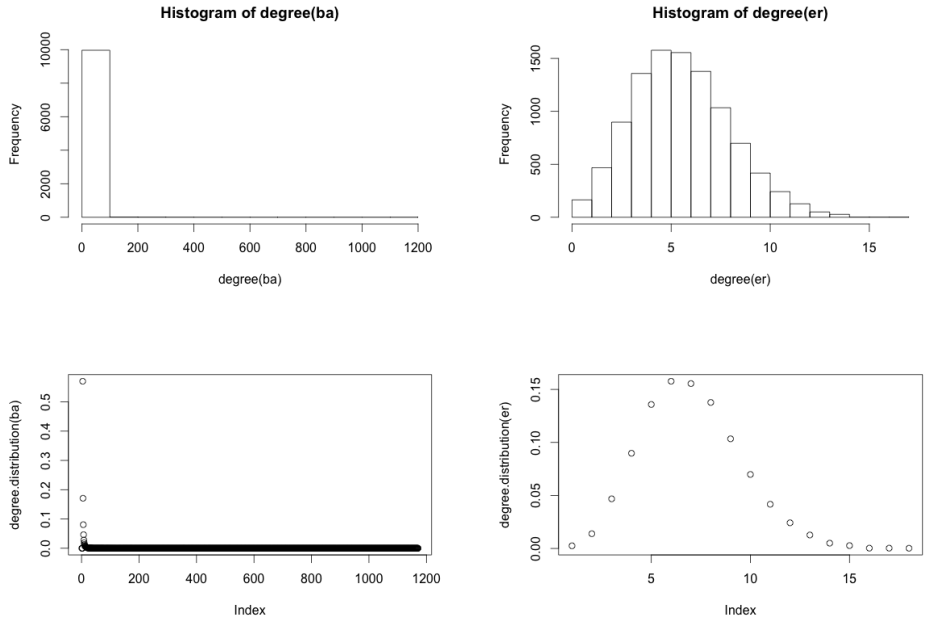
For `transitivity`

```
> ws <- watts.strogatz.game(1, 100, 4, 0.05)
```

```
> transitivity(ws)
[1] 0.5466147
> p_hat <- ecount(ws)/(vcount(ws)*(vcount(ws)-1)/2)
> p_hat
[1] 0.0808
> er <- erdos.renyi.game(100, p_hat)
> transitivity(er)
[1] 0.0830313
```

For `degree` and `degree.distribution`

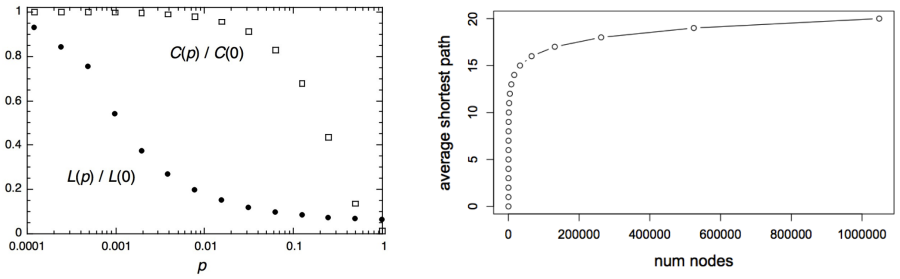
```
> g <- graph.ring(10)
> plot(g)
> degree(g)
 [1] 2 2 2 2 2 2 2 2 2 2

> 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 Your task

Your task is to reproduce these graphs introduced in our first lecture.



(a) (b)

That is, your task is to: (a) plot the clustering coefficient and the average shortest-path as a function of the parameter p of the WS model, and (b) plot the average shortest-path length as a function of the network size of the ER model.

Notice that in order to include both values — average shortest path and clus-

tering coefficient — in the same figure in plot (a), the clustering coefficient and the average shortest-path values are normalized to be within the range $[0, 1]$. This is achieved by dividing the values by the value obtained at the left-most point, that is, when $p = 0$.

In case of plot (b), you will have to experiment with appropriate values of p which may depend on the parameter n. You will notice that for large values of n your code may take too long, compute values for n that are reasonable for you. Also, make sure that you chose values for p that result (with high probability) in connected graphs. To achieve this, you can use a result from [1] stating:

- If $p < \frac{(1-\epsilon)\ln n}{n}$ then a graph in $G(n, p)$ will almost surely contain isolated vertices, and thus be disconnected
- If $p > \frac{(1+\epsilon)\ln n}{n}$ then a graph in $G(n, p)$ will almost surely be connected

4 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 (1 or 2 pages) describing your results. The formats accepted for the report are, in principle, pdf, Word, OpenOffice, and Postscript. You also have to hand in the source code of your implementations.

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

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

References

[1] Paul Erdős and A Rényi. On the evolution of random graphs. *Publ. Math. Inst. Hungar. Acad. Sci*, 5:17–61, 1960.