Complex Networks to Analyze Public Transportation Systems using R Version 2.0

Keiko Verônica Ono Fonseca
Marcelo de Oliveira Rosa
Ricardo Lüders

September 10, 2019

Abstract

Complex networks is a useful tool to analyze graph structures presenting some particular statistical behaviors. One of these structures is the public transportation system (PTS) of major worldwide cities. In our case, we focused on studying Brazilian PTS like São Paulo and Curitiba. To assist our students, we wrote this handout in order to give them a quick access to tools like `R` (a powerful statistical tool) and `igraph` (a package that deals with Complex Networks). In a short period of time they will be able to (hopefully) understanding the possibilities that these tools can offer.

Keiko Verônica Ono Fonseca
Marcelo de Oliveira Rosa
Ricardo Lüders

Contents

Chapter 1

Introduction

Mathematically, a complex network is a big graph with some particular statistical behavior. As a graph, it is formed by a group of vertices (or nodes) whose edges (or links) are set according to some behavior between those vertices. For example, to map public transportation systems (or PTS's), each vertex could represent a bus stop while the existence of bus lines traversing two given bus stops would determine the existence of a link between them.

Therefore, the essential part of modeling a complex network involves establishing the meaning of the vertices and the formation rule for creating edges between these vertices. Sometimes it is possible to build meaningless complex networks due to inconsistent understanding of what should be vertices and edges.

Once a good representation is conceived, we can use tools like `R` and `igraph` to extract numeric results to answer questions raised from the model like identifying critical paths or critical locations in a network, for example.

This work was written to be used after a lecture on complex networks. So we assumed that you had such a lecture previously and now is ready to work with such networks.

Chapter 2

R for students

R is a computer tool to perform statistical analysis. It is a free software that can be installed on several operational systems (Linux, Windows, MacOS). It has a interpreted programming language (different from compiled programming language as C, for example).

The normal package that you download from `https://www.r-project.org/` comes with a minimal set of libraries along with a interpreter shell (shell is a computer software that reads commands you digit, interpret them, execute them, and show their results). Alternatively, it is possible to find some R's implementations that uses a graphic interface to help the statistical analysis you want to perform. One of these graphical interfaces is `RStudio`. It can be downloaded from `https://www.rstudio.com/` (you still need to have R downloaded and installed in your computer before running `RStudio` due to ownership issues).

Below we list a few useful commands that you can use directly from the shell:

- What can I do?

  The function `help.start()` shows html-based information that helps you to take the first steps in R. Once you get to know the basic functions, you can use `?<function>` to have information about a particular function in R (named `<function>`).

  There is also the function `demo()` to guide you through some examples of statistical analysis that you can perform on R.

- Quitting the shell

  The first basic command you have in R is quitting the shell after having used it. Use `q()`. After that, R will ask you if you want to save all the variables you have now (to list them, use `ls()` in a single file that you can load latter).

- Listing all variables in use

Variables are created on R in similar way as in general computer languages (as C or Pascal). For example:

```
a = 0
```

or

```
a ←0
```

means that a variable a will be created and receive the value 0.

To list all the variable you have currently available, you can use the function ls().

- Removing one variable

  The function rm() is used to remove a variable. For example, to remove a variable named a, you write

  ```
  rm(a)
  ```

  If you want to remove all the variables, you write

  ```
  rm(ls())
  ```

  that means you need to list all the variable first in order to remove each of them, individually.

- Seeing the content of a variable

  In R, you just need to write the name of the variable to see its content.

- Running a custom code that you've developed

  Instead of simply typing all functions in R' shell, you can put all your functions in a form of a R's code in a text file. Normally the name of the file carries the extension .R (although you are not required to add it to your file).

  Let us assume you typed a bunch of lines in your favorite text editor and you saved in a file name /home/user/my_amazing_code.R. To run it in R' shell, simply type:

  ```
  source(/home/user/my_amazing_code.R)
  ```

  Any error or warnings will refer to a line in this file (to help you get rid of them).

- Setting up a working directory (for file operations, like reading or writing a file).

  Use function setwd() as:

  ```
  setwd('dir1/dir2/')
  ```

Chapter 3

Complex Networks on R

Here we want to describe how to build complex networks and calculate metrics on R.

3.1 Preparing the land

As mentioned before, R has several libraries (named packages in R) to expand its statistical analysis abilities. In case of Complex Networks, we will use a library called `igraph`, which can be installed through the following steps:

1. At R' shell, write:

```
install.packages(igraph)
```

2. R will ask you to choose a server around the world to download this package. Choose one near you to speed up the process;
3. Depending on the package, R has to download other dependent packages, or simply install the one you choose.
4. After that you will have the package installed on your system.

To use the package you need to write

```
require(igraph)
```

in order to have it loaded. It is also possible to call `library(igraph)`, however it is recommended to use `require()` because this function is suitable in subroutines or custom functions that you may build in the future.

3.2 Creating a graph

Formally a graph $G(V, E)$ is defined from two mathematical sets: V is the set of vertices, and E is the set of edges. An edge is an ordered pair of two elements belonging to V (or $(a, b)|a, b \in V$).

Another way to mathematically conceive a graph is from its adjacent matrix A. Be N the number of elements in V, A is a $N \times N$ (squared) matrix whose elements $A_{i,j} = 1$ if there is an edge $(V_i, V_j) \in E$, or $A_{i,j} = 0$ otherwise.

If A is symmetric ($A_{i,j} = A_{j,i}$), then the graph is undirected, which means there is no difference preferential order in taking edges (V_i, V_j) or (V_j, V_i). Otherwise the graph is a directed graph.

Adding weight to graph edges can be done by using a weighted version of A: instead of setting $A_{i,j} = 1$, we can use a different value representing the weight we want to set to such an edge. We should remember that setting $A_{i,j} = 0$ means lack of any connection between V_i and V_j.

In `R`, there are several ways to create a graph. We must remember that we need to create a variable that a graph and all information related to it.

3.2.1 From data frame

Data frame is a `R`' structure that resembles a table: each line could be seem as a different object in a set, and each column describes one of its characteristics. `R` has functions to directly work on it (for example, statistically summarizing it).

Applying `graph_from_data_frame()` over a data frame (or two - see `R`'s help for this function), it is possible to create a simple graph (we will use PTS entities in the examples to give a context to them):

```
df1 ←data.frame(
   from = c('STOP1', 'STOP2', 'STOP3', 'STOP4', 'STOP5',
            'STOP6', 'STOP7', 'STOP8', 'STOP9', 'STOP10'),
   to = c('STOP2', 'STOP3', 'STOP4', 'STOP5', 'STOP6',
            'STOP7', 'STOP8', 'STOP9', 'STOP10', 'STOP1'))
line_1 ←graph_from_data_frame(df1)
line_1
summary(line_1)
plot(line_1)
```

As you can see, we created a directed graph representing the bus stops in a line (we named this graph as `line_1`) with 10 bus stops following the direction that its vehicles traverse them.

Additionally we used `plot()` function to get a graphical representation of it where we see the graph vertices represented as circles and the ordered

edges as arrows. Typing the name of the graph variable or using `summary` function allows you to have a brief information about this graph.

In this example you can also see that we declared a data frame with two variables (`from` and `to`) since `graph_from_data_frame()` requires the presence of these column names in the data frame, so it can understand as the beginning and ending of the edges.

The function `c()` declares a vector in `R`. In this example, we are declaring two vectors of the same size, both containing strings.

To get more detailed information about the graph, we can type the following functions:

```
vcount(line_1)
ecount(line_1)
V(line_1)
V(line_1)$name
E(line_1)
as_adj(line_1)
line_1 ←graph_from_data_frame(df1, directed = FALSE)
E(line_1)
as_adj(line_1)
plot(line_1)
ends(line_1, es = E(line_1))
E(line_1)[from('STOP1')]
ends(line_1, es = E(line_1)[from('STOP1')])
ends(line_1, es = 5)
V(line_1)
V(line_1)[1]$name ←'START_STOP'
plot(line_1)
V(line_1)['START_STOP']$name ←'STOP1'
plot(line_1)
```

Functions `E()` and `V()` respectively extract the edges and vertices of a given graph variable. In this new example, we showed how to construct an undirected graph using the same data frame. Graphically we see a change in the edges, since they have no orientation anymore.

Both adjacent matrices were obtained by using function `as_adj()`. Note how naturally this graph representation indicates the relationship between the graph vertices.

Function `ends()` extracts one or more edges from a given graph variable while `E()[from()]` (and, or course `E()[to()]`) retrieves edges having a specific starting vertex (or a ending vertex).

Finally, it is possible to change parts of a given graph using the attribute operator `<-`. In this example, we changed the name of a bus stop and then reverted the operation.

3.2.2 A simple PTS

Using the result from previous examples, we are now capable of representing a simple PTS as a graph in R. In our example, we are considering 4 bus lines (it could be metro lines or whatever public transportation system we want).

```
df2 ←data.frame(
   from = c('STOP1', 'STOP11', 'STOP12', 'STOP6', 'STOP13',
            'STOP14', 'STOP15', 'STOP7', 'STOP16', 'STOP17',
            'STOP18', 'STOP19'),
   to = c('STOP11', 'STOP12', 'STOP6', 'STOP13', 'STOP14',
            'STOP15', 'STOP7', 'STOP16', 'STOP17', 'STOP18',
            'STOP19', 'STOP1'))
line_2 ←graph_from_data_frame(df2, directed = FALSE)
df3 ←data.frame(
   from = c('STOP11', 'STOP20', 'STOP10', 'STOP19', 'STOP21',
            'STOP22', 'STOP23', 'STOP24', 'STOP25', 'STOP2'),
   to = c('STOP20', 'STOP10', 'STOP19', 'STOP21', 'STOP22',
            'STOP23', 'STOP24', 'STOP25', 'STOP2', 'STOP11'))
line_3 ←graph_from_data_frame(df3, directed = FALSE)
df4 ←data.frame(
   from = c('STOP1', 'STOP2', 'STOP25', 'STOP30', 'STOP31',
            'STOP32', 'STOP33', 'STOP34'),
   to = c('STOP2', 'STOP25', 'STOP30', 'STOP31', 'STOP32',
            'STOP33', 'STOP34', 'STOP1'))
line_4 ←graph_from_data_frame(df4, directed = FALSE)
all_lines = line_1 + line_2 + line_3 + line_4
```

In this new example, we added three different lines to form a 4-line PTS. Note that some bus stops are shared among the four lines. In terms of graphs, we simply join the set of vertices (i.e. $V_{\texttt{line\_1}} \cup V_{\texttt{line\_2}} \cup V_{\texttt{line\_3}} \cup V_{\texttt{line\_4}}$) and the set of edges ($E_{\texttt{line\_1}} \cup E_{\texttt{line\_2}} \cup E_{\texttt{line\_3}} \cup E_{\texttt{line\_4}}$).

This kind of 'adding' approach leads to a important representation of PTS as complex network named **L-space**. It helps analyzes how the buses are connected, identify the bus stops (or terminals) that are used by most of the lines (possible point of line congestion).

However, it is possible to verify that some edges disappeared after joining all four lines in order to form the variable `all_lines`. The missed edges were the ones that repetitively occurred in more than one bus line as a result of the union operation performed over the sets of graph edges from all the lines.

The next example allowed us to build a graph including these repetitive edges.

```
all_lines_stops ←union(
   V(line_1)$name, union(V(line_2)$name,
      union(V(line_3)$name, V(line_4)$name)))
```

```
df ←data.frame(
   from = c(ends(line_1, es = E(line_1))[,1],
            ends(line_2, es = E(line_2))[,1],
            ends(line_3, es = E(line_3))[,1],
            ends(line_4, es = E(line_4))[,1]),
   to = c(ends(line_1, es = E(line_1))[,2],
            ends(line_2, es = E(line_2))[,2],
            ends(line_3, es = E(line_3))[,2],
            ends(line_4, es = E(line_4))[,2]),
   color = c(rep('gray', ecount(line_1)),
            rep('yellow', ecount(line_2)),
            rep('red', ecount(line_3)),
            rep('blue', ecount(line_4))))
all_lines_multi ←graph_from_data_frame(df, directed = FALSE)
plot(all_lines_multi, edge.width = 4)
```

Additionally, we included new column to a data frame used to describe the graph called `color`. It is interpreted by `igraph` as we want to set an attribute called `color` to each edge of our graph. Particularly we set different colors for different bus lines (it is possible to set a color for each edge, if necessary, or one color for all edges). Moreover, it is possible to create any kind of attribute to each edge, and also to each vertex.

Finally we plotted a graph whose edges are thicker than the ones we plotted in the previous examples. Now clearly we observe that stops `STOP1`, `STOP2`, and `STOP25` have more than one edge connecting them.

Such a graph is known as **multi-graph**. In case of PTS, it is a direct representation of bus lines covering a city, for example. We name it as a **multi-graph L-Space** representation.

Once we have this multi-graph L-Space mapped into `R`, it is possible to quickly obtain L-Space representation by using `simplify()` function.

```
simplify(all_lines_multi)
all_lines
simplify(all_lines_multi) - all_lines
```

Clearly `all_lines_multi` $\subset$ `all_lines`.

Now we will build a new representation of PTS for the same group of 4 lines. It is called **P-Space** representation. First we need to form a complete graph of each bus line: it means that we create edges for all pair of vertices. In this case, we create connections between all pair of bus stops.

Next we add the complete graphs from all bus lines (as mentioned before) to create a P-Space graph. In the example below we used one of the functions provided by `igraph` to build complete graphs (`make_full_graph`). Since this function creates a graph without naming its vertices, we copied the vertex

names of a previous graph into this new one (for `igraph`, an element name is just a new attribute, so we can easily do that).

```
line_1_p ←make_full_graph(vcount(line_1))
V(line_1_p)$name ←V(line_1)$name
line_2_p ←make_full_graph(vcount(line_2))
V(line_2_p)$name ←V(line_2)$name
line_3_p ←make_full_graph(vcount(line_3))
V(line_3_p)$name ←V(line_3)$name
line_4_p ←make_full_graph(vcount(line_4))
V(line_4_p)$name ←V(line_4)$name
# %u% does the same as +
all_lines_p ←line_1_p %u% line_2_p %u% line_3_p %u% line_4_p
plot(all_lines_p)
```

Similar to the multi-graph L-Space, we can form a **multi-graph P-Space** by keeping the repetitive edges between a few pair of vertices. In the next example, we also added color to easily visualize the connection between lines.

```
all_lines_stops_p ←V(all_lines_p)
df_p ←data.frame(
  from = c(ends(line_1_p, es = E(line_1_p))[,1],
           ends(line_2_p, es = E(line_2_p))[,1],
           ends(line_3_p, es = E(line_3_p))[,1],
           ends(line_4_p, es = E(line_4_p))[,1]),
  to = c(ends(line_1_p, es = E(line_1_p))[,2],
           ends(line_2_p, es = E(line_2_p))[,2],
           ends(line_3_p, es = E(line_3_p))[,2],
           ends(line_4_p, es = E(line_4_p))[,2]),
  color = c(rep('gray', ecount(line_1_p)),
           rep('yellow', ecount(line_2_p)),
           rep('red', ecount(line_3_p)),
           rep('blue', ecount(line_4_p))))
all_lines_p_multi ←graph_from_data_frame(
   df_p, directed = FALSE)
plot(all_lines_p_multi, edge.width = 4)
```

The importance of P-Space is that we are able to determine, for example, the number of transfers between bus lines to reach destinations, for example. We will see it latter.

3.2.3 Creating synthetic networks

As we seen at the end of the previous section, we can create synthetic graphs to assist you. We will present now a few possible graphs that are useful as

reference for studying real-world complex networks.

It is important to mention that the difference between complex networks and graphs is that while the latter is related to mathematical and computing aspects, generally dealing with synthetic models and their properties, the former is related to big real-world models and similarities between these models. Nowadays, analysis of social media is being done using huge graphs whose statistical behavior may be similar to neuron connections in human brain, for example.

```
# random network
g_random ←erdos.renyi.game(50, 1/10)
plot(g_random, edge.width = 4)
# scale-free/barabasi-albert model
g_scale_free ←barabasi.game(50, directed = FALSE)
plot(g_scale_free, edge.width = 4)
# small-world
g_small_world ←sample_smallworld(1, 50, 5, 0.05)
plot(g_small_world, edge.width = 4)
```

In complex network studies, we have three types of networks that are useful for comparison with real-world networks. First we have **random networks**, whose edges between pair of vertices are randomly created meaning that there is no preferential attachment or relationship between graph vertices.

Next we have **scale-free networks**. In such networks, a few vertices are present in most of graph edges, acting as a hub for the rest of the vertices. If we consider paths built between all pair of vertices, most of them will have one of these few hubs as intermediate vertices.

Finally, **small-world networks** map networks where a large number of vertices are connected to other vertices and these ones are mutually connected. This situation is found in social relationships where you have two friends, for example, that know each other. Other simplified idea is that every one can be connected to each other considering friend of friend of friends, and so on (Six degree of Kevin Bacon).

We reinforce that these are only theoretical networks and that real-world networks may resemble them but do not completely follow their statistical behavior. Also, it is important to mention that they are not the only useful complex networks available.

3.3 Metrics

Metrics are a set of numbers that measures one or more characteristics of a graph or complex network. We will focus on a few metrics (the basic ones) in order to assist your understanding about a given graph.

3.3.1 Degree

The first measurement found in literature is degree (k_i where i is the index of the graph vertex). It accounts the number of edges connecting the i^{th} vertex. In case of a directed graph, you can have degree associated to edges come in or get out the the i^{th} vertex.

Assuming a PTS represented in L-Space, $k_i/2$ roughly measures the number of bus lines supported by the i^{th} bus stop.

```
degree(g_random)
degree(g_scale_free)
degree(g_small_world)
mean(degree(g_random))
mean(degree(g_scale_free))
mean(degree(g_small_world))
V(g_random)$size = degree(g_random) * 2
plot(g_random, edge.width = 4)
V(g_scale_free)$size = degree(g_scale_free) * 2
plot(g_scale_free, edge.width = 4)
V(g_small_world)$size = degree(g_small_world) * 2
plot(g_small_world, edge.width = 4)
```

In this example, we calculated the degree of all vertices. Additionally we plotted some synthetic graphs changing the radius of the circle that represents each vertex according to the magnitude of its degree. It help us identifying the bus stops that sustain large number of bus lines, for example.

While in scale-free networks we observe a few vertices with large k_i, vertices in random networks naturally present similar degree (low variance).

3.3.2 Shortest path length

The shortest path length (or path length, symbolic defined as $l_{i,j}$) is the minimum number of edges used to connect two given vertices (V_i and V_j). For L-Space PTS's, it indicates the number of bus stops needed to traverse two given stops (disregarding the number of bus transfers involved).

In the following example, we present the main `R` functions that you can use to calculate this metric.

```
mean_distance(g_random)
mean_distance(g_scale_free)
mean_distance(g_small_world)
diameter(g_random)
diameter(g_scale_free)
diameter(g_small_world)
distances(all_lines)[1:10, 1:10]
distances(all_lines, 'STOP1', 'STOP3')
```

```
result ←shortest_paths(all_lines, 'STOP14', 'STOP31',
                       output = 'epath')
all_lines_path ←all_lines
E(all_lines_path)$color = 'gray'
E(all_lines_path)$width = '2'
E(all_lines_path)[result$epath[[1]]]$color = 'red'
E(all_lines_path)[result$epath[[1]]]$width = '4'
plot(all_lines_path)
rm(all_lines_path)
```

Note that we can calculate $\langle l \rangle$ (or the average of all $l_{i,j}$'s), a squared matrix containing $l_{i,j}$ for all graph vertices, the graph diameter (longest shortest path in a given graph), and the edges connecting two given vertices.

To visualize a given shortest path (formed by a sequence of edges, but we can also obtain a sequence of vertices) connecting two given vertices, we added default attributes to all the edges of a graph, and set different values for these attributes to the ones belonging that shortest path.

3.3.3 Network centrality

Centrality metrics are useful to identify the most important vertices to the network. There are several ways to measure such importance. Betweenness and closeness centrality are two metrics based on the path length between all pair of vertices.

Closeness centrality ranks all vertices according to how close are them to the rest of the network. Therefore the closeness centrality of the i^{th} graph vertex is calculated as:

$$C_i = \frac{1}{\sum_j l_{i,j}} \tag{3.1}$$

In a PTS, it can be used to identify the level of reachability of a given bus stop in terms of how fast it takes to a user departing from that bus stop to reach any other bus stop.

Betweenness centrality ranks all vertices according to the number of shortest paths having them as intermediate vertices. It measures the importance of a bus stop as having large number of paths passing through it. It is computed by:

$$B_i = \sum_{i \neq j \neq k} \frac{\text{path}_{j,k}(i)}{\text{path}_{j,k}} \tag{3.2}$$

where $\text{path}_{j,k}$ corresponds to the number of shortest paths from V_i to V_j, and $\text{path}_{j,k}(i)$ corresponds to the number of shortest paths from V_i to V_j passing through V_i. Note that B_i is a sum of fractions since both $\text{path}_{j,k}$ and $\text{path}_{j,k}(i)$ can only result in zero or one.

In PTS, betweenness helps to identify bus stops that can receive large amount of users since they belong to most of the shortest paths that can be derived from a network. A failure in such bus stops could result in difficulties to PTS managers, for example.

Both metrics are very computing intensive since they require that all shortest paths be determined. In the example we present below, we calculated the betweenness and closeness centralities of a multi-graph L-Space network.

```
betweenness(all_lines_multi)
closeness(all_lines_multi)
all_lines_centrality ←all_lines_multi
V(all_lines_centrality)$size ←
   betweenness(all_lines_centrality)/5
plot(all_lines_centrality, edge.width = 4)
all_lines_centrality ←all_lines_p_multi
V(all_lines_centrality)$size ←
     closeness(all_lines_centrality)*1000
plot(all_lines_centrality, edge.width = 4)
```

Since they are calculated at vertex level, we used their results to plot all vertices with different sizes according to their centrality values (we had to apply some gain to the values in order to have a good visual effect).

3.3.4 Clustering coefficients

Clustering coefficient (or network transitivity) evaluates how connection tightness between neighbor vertices. In social networks, assume that I (vertex) have N friends (N vertices and N edge): this metric measures how many of my friends are friends to each other. Therefore we have values measuring if all of them are friends to each other (maximum clustering coefficient) to values measuring if none of my friends have friendship among them (minimum clustering coefficient).

In theoretical small-world networks, we have high clustering coefficient, since there is a high probability of neighbor vertices been connected through edges.

The literature presents two versions of calculating the clustering coefficient of a network: the global and the local version. The former retrieves a value representing the network metric while the later is calculated at each network vertex (at the end, we can apply descriptive statistic to get the distribution of this metric across the network).

```
transitivity(g_random, type = 'global')
transitivity(g_smallworld, type = 'global')
transitivity(g_scale_free, type = 'global')
```

```
mean(transitivity(g_scale_free, type = 'local')
all_lines_transitivity ←all_lines
V(all_lines_transitivity)$size ←
  transitivity(all_lines_transitivity, 'local') * 100
plot(all_lines_transitivity, edge.width = 4)
```

3.3.5 Network Modularity

This complex network metric attempts to detect community structures in a given network. Members (vertices) of such communities should be highly connected to each other or present some characteristic that make them part of a community. Therefore, there are different algorithms to determine such membership.

Next example we explore two of them. Details on how they are implemented, look for `igraph` help (the first one is very memory demanding, so be careful in using it in medium-to-large networks, as the ones we will see later).

```
my_members ←cluster_edge_betweenness(all_lines)
length(my_members)
plot(my_members, all_lines)
membership(my_members)
modularity(my_members)

my_members ←cluster_walktrap(all_lines)
length(my_members)
plot(my_members, all_lines)
membership(my_members)
modularity(my_members)
```

3.3.6 Weighted Networks

All previous analysis consisted on assuming the weight of all edges as one. It means that metrics as $l_{i,j}$ counts number of edges between V_i and V_j, regardless the significance of those edges. In communications, it means the number of hops between those vertices.

In PTS, it is interesting to take the real distance between bus stops into account in order to evaluate how far a passenger has to travel between two given bus stops if he/she choose to take the shortest path (assuming wise choices).

Graphs in `R` can include edge weights as edge attributes. Particularly if we set an attribute named `weight`, `igraph` takes it into computing metrics. We will see later (in a real-world problem) how to manage weighted complex networks in `R`.

Chapter 4

Maps on R

Similar to the package `igraph`, there are several packages in `R` that deal with maps. Here we are going to use the package `ggmap`. It can access maps from Google Maps, OpenStreetMap, and Stamem Maps. Additionally, it uses low level plotting interfaces from package `ggplot2`, which we will use to add pictograph information over the maps.

4.1 Installing map support

Similar to what we did to package `igraph` in section 3.1, we need to install package `ggmap`. So, type:

```
install.packages(ggmap)
```

choose your server to download such a package and wait for a moment. After having it installed, type:

```
require(ggmap)
```

4.2 Plotting a few maps

The following example allows you to view different maps from Stockholm. `get_map` is a wrapper function that internally deals with different source of maps in a unified programming interface. This is the first step to get useful maps plotted.

```
require(ggmap)
my_stockholm ←get_map(location = 'stockholm',
   maptype = 'roadmap')
ggmap(my_stockholm)
my_stockholm ←get_map(location = 'stockholm',
   zoom = 13)
```

```
ggmap(my_stockholm)
my_stockholm ←get_map(location = 'stockholm',
   maptype = 'watercolor', zoom = 13, source = 'stamen')
ggmap(my_stockholm)
```

Important: Since July 16th, 2018, Google requires an API key to use their map service. So when some functions of `ggmap` access Google Maps API, it requires such key. It is their right to ask for payments (or not) to give us access to their service. There was one free alternative called OpenStreeMap embedded into `ggmap` but it has been deactivated since OpenStreetMap owners have reinforced the only user to their maps is their own site. The only remained option is Statmem, another free alternative in `ggmap`. The following code is a translation of the last `R` code from Google Maps to Stamen.

```
require(ggmap)
city_center ←c(18.068351, 59.334591)
city_region ←c(
   left = city_center[1] - 0.05,
   top = city_center[2] + 0.03,
   right = city_center[1] + 0.05,
   bottom = city_center[2] - 0.03)
my_stockholm ←get_map(city_region,
   zoom = 12)
ggmap(my_stockholm)
my_stockholm ←get_map(city_region,
   zoom = 13)
ggmap(my_stockholm)
my_stockholm ←get_stamenmap(city_region,
   maptype = 'watercolor', zoom = 13)
ggmap(my_stockholm)
```

We can also mark places directly on a plotted map using functions from `ggplot2`.

```
my_stockholm ←get_map(location = c(18.150845, 59.349610),
  zoom = 12)
df_map ←data.frame(name = c('hotel', 'kth'),
                   lat = c(59.365165, 59.353537),
                   lon = c(18.239665, 18.065407))
ggmap(my_stockholm, extent = 'device') +
  geom_path (aes(x = lon, y = lat), data = df_map,
    size = 2, color = 'blue') +
  geom_point(aes(x = lon, y = lat), data = df_map[1, ],
    size = 3, color = 'black') +
  geom_point(aes(x = lon, y = lat), data = df_map[2, ],
    size = 3, color = 'red')
```

Using Stamen:

```
city_center ←c(18.150845, 59.349610)
city_region ←c(
  left = city_center[1] - 0.10,
  top = city_center[2] + 0.04,
  right = city_center[1] + 0.10,
  bottom = city_center[2] - 0.04)
my_stockholm ←get_map(city_region,
  zoom = 13)
df_map ←data.frame(name = c('hotel', 'kth'),
                   lon = c(18.239665, 18.065407),
                   lat = c(59.365165, 59.353537))
ggmap(my_stockholm, extent = 'device') +
  geom_path (aes(x = lon, y = lat), data = df_map,
    size = 2, color = 'blue') +
  geom_point(aes(x = lon, y = lat), data = df_map[1, ],
    size = 3, color = 'black') +
  geom_point(aes(x = lon, y = lat), data = df_map[2, ],
    size = 3, color = 'red')
```

4.3 Real data: Curitiba

The following example uses real data from Curitiba, Brazil in order to plot information about its PTS (data from 2016). We converted from JSON files provided by URBS (municipal company that organized the public transportation in Curitiba) into a list of lists.

The main list consists on information about each Curitiba's bus line (name of the line, an id and a category assigned to that). For each entry in this list, there is a list of possible bus directions, which comprises a list of bus stops ordered according to the direction that a bus in that direction should traverse.

Therefore we have:

`list_of_lines{ list_of_directions {list_of_bus_stops} }`

Such information is stored in a file (`all_bus_stops.R`, provided during this lecture). Therefore we use it to populated `R`' data frames in order to place all city bus stops over a map of the city (from Google Maps).

Additionally, we added different colors to each bus stop according to the category of its bus line (in Curitiba, all bus lines are categorized according to their "function" to the system. Detailed information of each function in the following `R` code can be obtained by typing `?function`). Note that some

bus stops may belong to different bus line categories, so we will pick up just one of them for graphical purposes.

```
load('all_bus_info.R')

# get bus line information
df_lines ←data.frame(index = numeric(length(all_bus_info)),
                     code = character(length(all_bus_info)),
                     name = character(length(all_bus_info)),
                     cat = character(length(all_bus_info)),
                     stringsAsFactors = FALSE)
for (i in 1:length(all_bus_info)) {
  df_lines$index[i] ←i
  df_lines$code[i] ←all_bus_info[[i]][[1]]
  df_lines$name[i] ←all_bus_info[[i]][[2]]
  df_lines$cat[i] ←all_bus_info[[i]][[3]]
}

# get bus stop information
df_all_stops ←data.frame(num = character(0),
                         lat = numeric(0),
                         lon = numeric(0),
                         group = character(0),
                         cat = character(0),
                         stringsAsFactors = FALSE)
for (i in 1:length(all_bus_info)) {
  num_of_directions ←length(all_bus_info[[i]][[4]])
  if (num_of_directions==0)
    next

  a_line_code ←all_bus_info[[i]][[1]]

  for (j in 1:num_of_directions) {
    total_stops ←nrow(all_bus_info[[i]][[4]][[j]][[2]])
    df_sub ←data.frame(num = character(total_stops),
                       lat = numeric(total_stops),
                       lon = numeric(total_stops),
                       group = character(total_stops),
                       cat = character(total_stops),
                       stringsAsFactors = FALSE)
    df_sub$num ←all_bus_info[[i]][[4]][[j]][[2]]$NUM
    df_sub$lat ←all_bus_info[[i]][[4]][[j]][[2]]$LAT
    df_sub$lon ←all_bus_info[[i]][[4]][[j]][[2]]$LON
    df_sub$group ←all_bus_info[[i]][[4]][[j]][[2]]$GRUPO
```

```
    df_sub$cat ←df_lines[df_lines$code==a_line_code,]$cat

    df_all_stops ←rbind(df_all_stops, df_sub)
  }
}

# removing repetition
df_all_stops ←df_all_stops[!duplicated(df_all_stops$num), ]

# plotting
city_center = c(mean(df_all_stops$lon), mean(df_all_stops$lat))
my_curitiba_v11 ←get_map(location = city_center, zoom = 11)
ggmap(my_curitiba_v11, extent = 'device') +
  geom_point(aes(x = lon, y = lat, color = factor(cat)),
  data = df_all_stops)
```

For Stamen, the last 4 lines can be replaced to:

```
a_delta = 0.001
city_region ←c(
       left = min(df_all_stops$lon)-a_delta,
       top = max(df_all_stops$lat)+a_delta,
       right = max(df_all_stops$lon)+a_delta,
       bottom = min(df_all_stops$lat)-a_delta)
my_curitiba_v11 ←get_map(
  location = city_region, zoom = 12)
ggmap(my_curitiba_v11, extent = 'device') +
  geom_point(aes(x = lon, y = lat, color = factor(cat)),
  data = df_all_stops)
```

4.3.1 Creating a graph for Curitiba

Once we have data from a real PTS, we will now create a graph that incorporates some of geographical information of a city (Curitiba, Brazil). We present two alternatives (out of other possible solutions) to do that. All they relied on previous computed data using earlier examples.

In this first `R` code, we got the bus stop information (only ids and edges) from bus lines (one or more directions) into a subgraph, and added all these subgraphs into a L-space graph of the city. At the end, we added geographic information (lat/lon of the bus stops, and geographic distance between them) to plot and obtain a weighted graph of the city (distance as edge weights).

```
require(geosphere)
```

```
curitiba_l ←make_empty_graph(directed = FALSE)
df_empty ←data.frame(from = character(0),
                     to = character(0))

for (i in 1:length(all_bus_info)) {
  num_of_directions ←length(all_bus_info[[i]][[4]])
  if (num_of_directions==0)
    next

  df_stops_per_line ←df_empty
  for (j in 1:num_of_directions) {
              paste(i)
    a_df ←all_bus_info[[i]][[4]][[j]][[2]]

    total_stops ←nrow(a_df)
    df_sub ←data.frame(from = a_df$NUM[1:nrow(a_df) - 1],
                       to = a_df$NUM[-1])

    df_stops_per_line ←rbind(df_stops_per_line, df_sub)
  }

  curitiba_l ←curitiba_l +
    graph_from_data_frame(df_stops_per_line, directed = FALSE)
}

# feeding information of each vertex/bus stop
V(curitiba_l)[order(V(curitiba_l)$name)]$lat ←
  df_all_stops[order(df_all_stops$num),]$lat
V(curitiba_l)[order(V(curitiba_l)$name)]$lon ←
  df_all_stops[order(df_all_stops$num),]$lon
V(curitiba_l)[order(V(curitiba_l)$name)]$group ←
  df_all_stops[order(df_all_stops$num),]$group

# feeding distance (in meters) between edges
bus_stop_connection ←as_edgelist(curitiba_l)
idx_1 ←vapply(bus_stop_connection[,1],
              function(key) { which(df_all_stops$num==key) },
              0)
idx_2 ←vapply(bus_stop_connection[,2],
              function(key) { which(df_all_stops$num==key) },
              0)
E(curitiba_l)$weight ←distHaversine(
  cbind(df_all_stops$lon[idx_1], df_all_stops$lat[idx_1]),
  cbind(df_all_stops$lon[idx_2], df_all_stops$lat[idx_2]))
```

The next example performs the same as the last one. However, instead of adding edge weights at the end of the code, it includes the edge weights during the process of obtaining all subgraphs.

Characteristics of `igraph` slow down its performance but the idea is just to show a different way to make a graph from real data.

```
curitiba_l ←make_empty_graph(directed = FALSE)
df_empty ←data.frame(from = character(0),
                     to = character(0))

for (i in 1:length(all_bus_info)) {
  num_of_directions ←length(all_bus_info[[i]][[4]])
  if (num_of_directions==0)
    next

  df_stops_per_line ←df_empty
  for (j in 1:num_of_directions) {
    a_df ←all_bus_info[[i]][[4]][[j]][[2]]
    total_stops ←nrow(a_df)
    df_sub ←data.frame(
      from = a_df$NUM[1:nrow(a_df) - 1],
      to = a_df$NUM[-1],
      weight = distHaversine(
        cbind(a_df$LON[1:nrow(a_df) - 1],
              a_df$LAT[1:nrow(a_df) - 1]),
        cbind(a_df$LON[-1],
              a_df$LAT[-1]))
    )
    df_stops_per_line ←rbind(df_stops_per_line, df_sub)
  }

  curitiba_l ←curitiba_l +
    graph_from_data_frame(df_stops_per_line, directed = FALSE)

  if(length(grep('_[[:digit:]]\\b',
                 edge_attr_names(curitiba_l)))) {
    E(curitiba_l)$weight ←
      apply(cbind(E(curitiba_l)$weight_1,
                  E(curitiba_l)$weight_2),
            1, mean, na.rm = TRUE)
    curitiba_l ←delete_edge_attr(curitiba_l, 'weight_1')
    curitiba_l ←delete_edge_attr(curitiba_l, 'weight_2')
  }
```

```
}

# feeding information of each vertex/bus stop
V(curitiba_l)[order(V(curitiba_l)$name)]$lat ←
  df_all_stops[order(df_all_stops$num),]$lat
V(curitiba_l)[order(V(curitiba_l)$name)]$lon ←
  df_all_stops[order(df_all_stops$num),]$lon
V(curitiba_l)[order(V(curitiba_l)$name)]$group ←
  df_all_stops[order(df_all_stops$num),]$group
```

4.3.2 Plotting bus line(s)

It is possible to get a graph from different parts of Curitiba's PTS. The next example shows how to create a graph from a specific bus line (`020`) and plot it over the city's map (we walk over the structure called `all_bus_info`) in order to find information about this bus line, and create a graph with it).

```
wanted_line_code = '020'

bus_line_graph_l ←make_empty_graph(directed = FALSE)
df_empty ←data.frame(from = character(0),
                     to = character(0))

for (i in 1:length(all_bus_info)) {
  num_of_directions ←length(all_bus_info[[i]][[4]])
  if (num_of_directions==0)
    next

  a_line_code ←all_bus_info[[i]][[1]]
  if (a_line_code==wanted_line_code)
  {
    df_stops_per_line ←df_empty
    for (j in 1:num_of_directions) {
      a_df ←all_bus_info[[i]][[4]][[j]][[2]]
      total_stops ←nrow(a_df)
      df_sub ←data.frame(from = a_df$NUM[1:nrow(a_df) - 1],
                         to = a_df$NUM[-1])

      df_stops_per_line ←rbind(df_stops_per_line, df_sub)
    }

    bus_line_graph_l ←
      graph_from_data_frame(df_stops_per_line, directed = FALSE)
```

```
    break;
  }
}

# feeding information of each vertex/bus stop
idx_1 <-vapply(V(bus_line_graph_l)$name,
               function(key) { which(df_all_stops$num==key) },
               0)
V(bus_line_graph_l)$lat <-df_all_stops$lat[idx_1]
V(bus_line_graph_l)$lon <-df_all_stops$lon[idx_1]
V(bus_line_graph_l)$group <-df_all_stops$group[idx_1]

# feeding distance (in meters) between edges
bus_stop_connection <-as_edgelist(bus_line_graph_l)
idx_1 <-vapply(bus_stop_connection[,1],
               function(key) { which(df_all_stops$num==key) },
               0)
idx_2 <-vapply(bus_stop_connection[,2],
               function(key) { which(df_all_stops$num==key) },
               0)
E(bus_line_graph_l)$weight <-distHaversine(
  cbind(df_all_stops$lon[idx_1], df_all_stops$lat[idx_1]),
  cbind(df_all_stops$lon[idx_2], df_all_stops$lat[idx_2]))

# plotting such a bus line
df_wanted_line_1 <-data.frame(lat = V(bus_line_graph_l)$lat,
                              lon = V(bus_line_graph_l)$lon)
df_wanted_line_2 <-data.frame(x = df_all_stops$lon[idx_1],
                              y = df_all_stops$lat[idx_1],
                              xend = df_all_stops$lon[idx_2],
                              yend = df_all_stops$lat[idx_2])

my_curitiba_v12 <-get_map(location = city_center, zoom = 12)
ggmap(my_curitiba_v12, extent = 'device') +
  geom_segment(aes(x = x, y = y, xend = xend, yend = yend),
               data = df_wanted_line_2) +
  geom_point(aes(x = lon, y = lat),
             data = df_wanted_line_1,
             color = 'red')
```

For Stamen, the last 7 lines can be replaced to:

```
a_delta = 0.001
city_region <-c(
       left = min(df_all_stops$lon)-a_delta,
```

```
       top = max(df_all_stops$lat)+a_delta,
       right = max(df_all_stops$lon)+a_delta,
       bottom = min(df_all_stops$lat)-a_delta)
my_curitiba_v12 <-get_map(
  location = city_region, zoom = 13)
ggmap(my_curitiba_v12, extent = 'device') +
  geom_segment(aes(x = x, y = y, xend = xend, yend = yend),
               data = df_wanted_line_2) +
  geom_point(aes(x = lon, y = lat),
             data = df_wanted_line_1,
             color = 'red')
```

Following the same ideas exposed in the last example, we can plot all bus lines of a specific category.

```
wanted_category = 'EXPRESSO'

bus_line_graph_l <-make_empty_graph(directed = FALSE)
df_empty <-data.frame(from = character(0),
                      to = character(0))

for (i in 1:length(all_bus_info)) {
  num_of_directions <-length(all_bus_info[[i]][[4]])
  if (num_of_directions==0)
    next

  a_wanted_category <-all_bus_info[[i]][[3]]
  if (a_wanted_category==wanted_category)
  {
    df_stops_per_line <-df_empty
    for (j in 1:num_of_directions) {
      a_df <-all_bus_info[[i]][[4]][[j]][[2]]
      total_stops <-nrow(a_df)
      df_sub <-data.frame(from = a_df$NUM[1:nrow(a_df) - 1],
                          to = a_df$NUM[-1])

      df_stops_per_line <-rbind(df_stops_per_line, df_sub)
    }

    bus_line_graph_l <-bus_line_graph_l +
      graph_from_data_frame(df_stops_per_line, directed = FALSE)
  }
}

# feeding information of each vertex/bus stop
```

```
idx_1 <-vapply(V(bus_line_graph_l)$name,
               function(key) { which(df_all_stops$num==key) },
               0)
V(bus_line_graph_l)$lat <-df_all_stops$lat[idx_1]
V(bus_line_graph_l)$lon <-df_all_stops$lon[idx_1]
V(bus_line_graph_l)$group <-df_all_stops$group[idx_1]

# feeding distance (in meters) between edges
bus_stop_connection <-as_edgelist(bus_line_graph_l)
idx_1 <-vapply(bus_stop_connection[,1],
               function(key) { which(df_all_stops$num==key) },
               0)
idx_2 <-vapply(bus_stop_connection[,2],
               function(key) { which(df_all_stops$num==key) },
               0)
E(bus_line_graph_l)$weight <-distHaversine(
  cbind(df_all_stops$lon[idx_1], df_all_stops$lat[idx_1]),
  cbind(df_all_stops$lon[idx_2], df_all_stops$lat[idx_2]))

# plotting such a bus line
df_wanted_line_1 <-data.frame(lat = V(bus_line_graph_l)$lat,
                              lon = V(bus_line_graph_l)$lon)
df_wanted_line_2 <-data.frame(x = df_all_stops$lon[idx_1],
                              y = df_all_stops$lat[idx_1],
                              xend = df_all_stops$lon[idx_2],
                              yend = df_all_stops$lat[idx_2])

ggmap(my_curitiba_v12, extent = 'device') +
  geom_segment(aes(x = x, y = y, xend = xend, yend = yend),
               data = df_wanted_line_2) +
  geom_point(aes(x = lon, y = lat),
             data = df_wanted_line_1,
             color = 'red')
```

4.3.3 Combining metrics and maps

Now we will overlap maps and complex network metrics in order to have visual understanding of Curitiba's PTS. Considering the metric degree (Section 3.3.1), we plotted filled circles around each bus stop whose size is directly related to the bus stop degree.

Additionally, we also varied the colors of these circles to facilitate the identification of level of vertex degrees across the city.

```
df_bus_stops <-data.frame(lat = V(curitiba_l)$lat,
```

```
                        lon = V(curitiba_l)$lon)

# remove 'simplify' to see what's happen!
ctba_degree <-degree(simplify(curitiba_l))

df_metrics <-df_bus_stops
df_metrics$value <-ctba_degree

my_break_function <-function(x) {
  seq(min(x), max(x), length.out = 6)
}
ggmap(my_curitiba_v11, extent = 'device') +
  geom_point(data = df_metrics,
             aes(x = lon, y = lat,
             size = value, color = value)) +
  scale_size_continuous(range = c(0, 3),
                        breaks = my_break_function) +
  scale_color_gradient(breaks = my_break_function) +
  guides(size = guide_legend(title = NULL),
         color = guide_legend(title = NULL))
```

Network centrality metrics like betweenness (Section 3.3.3) also can be viewed in the same way. The next example allows to see the betweenness of the city network when we discard edge weights.

Since betweenness calculation is quite computing demanding, wait for a moment until R finishes the job.

```
df_bus_stops <-data.frame(lat = V(curitiba_l)$lat,
                          lon = V(curitiba_l)$lon)

# choose: NA->unweighted NULL->weighted
ctba_betweeness <-betweenness(simplify(curitiba_l),
                              weight = NULL,
                              normalized = TRUE)

df_metrics <-df_bus_stops
df_metrics$value <-ctba_betweeness
ggmap(my_curitiba_v11, extent = 'device') +
  geom_point(data = df_metrics,
             aes(x = lon, y = lat,
                 size = value, color = value)) +
  scale_size_continuous(range = c(0, 3),
                        breaks = my_break_function) +
  scale_color_gradient(breaks = my_break_function) +
  guides(size = guide_legend(title = NULL),
```

```
        color = guide_legend(title = NULL))
```

We can also use what is called thermal graphs in order to perceive variation of a metric "across the city". Package `ggplot2` have a way to automatically do that using `geom_density_2d()`.

However such a function does not take into account weights associated to the data provided to it. So we need to build a function to do that.

In the next example, we built such a function (it builds Gaussian mixtures function according to sampling data and its weight). It also requires package `MASS` installed in your system (by now you should know how to install any package).

```
require(MASS)

# Copied from MASS::kde2d and modified Ort Christoph
# https://stat.ethz.ch/pipermail/r-help/2006-June/107405.html
kde2d.weighted ←function (x, y, w, h, n = 25,
                          lims = c(range(x), range(y))) {
  nx ←length(x)
  if (length(y) != nx)
    stop("data␣vectors␣must␣be␣the␣same␣length")

  gx ←seq(lims[1], lims[2], length = n) # gridpoints x
  gy ←seq(lims[3], lims[4], length = n) # gridpoints y
  if (missing(h))
    h ←c(bandwidth.nrd(x), bandwidth.nrd(y));
  if (missing(w))
    w ←numeric(nx)+1;
  h ←h/4
  # distance of each point to each grid point in x-direction
  ax ←outer(gx, x, "-")/h[1]
  # distance of each point to each grid point in y-direction
  ay ←outer(gy, y, "-")/h[2]
  # z is the density
  z ←(matrix(rep(w,n), nrow=n, ncol=nx, byrow=TRUE) *
        matrix(dnorm(ax), n, nx)) %*%
    t(matrix(dnorm(ay), n, nx))/(sum(w) * h[1] * h[2])

  return(list(x = gx, y = gy, z = z))
}

df_bus_stops ←data.frame(lat = V(curitiba_l)$lat,
                         lon = V(curitiba_l)$lon)

# to correct a problem with our DATA
```

```
# (different bus stops with same lat/lon)
df_bus_stops$lat ←jitter(df_bus_stops$lat)

# choose: NA->unweighted NULL->weighted
ctba_betweeness ←betweenness(simplify(curitiba_l),
                             weight = NULL,
                             normalized = TRUE)

df_metrics ←df_bus_stops
df_metrics$value ←ctba_betweeness

a_density ←kde2d.weighted(df_metrics$lon,
                          df_metrics$lat,
                          df_metrics$value,
                          n=100)
df_density ←data.frame(
  expand.grid(x = a_density$x,
              y = a_density$y),
              z = as.vector(a_density$z))

ggmap(my_curitiba_v11, extent = 'device') +
  geom_contour(data = df_density,
               aes(x = x, y = y, z = z), binwidth = 5) +
  stat_contour(data = df_density,
               aes(x = x, y = y, z = z,
                   fill = ..level..,
                   alpha = ..level..),
               geom = 'polygon') +
  scale_fill_gradient(low = 'green',
                      high = 'red',
                      guide = FALSE) +
  scale_alpha(range=c(0, 0.30), guide = FALSE)
```

Chapter 5

Conclusion

The purpose of this handout is to provide information and basic concepts to anyone interesting on using complex network approach to model systems or process using tools like `R` and `igraph`, particularly those researching subjects involving public transportation system or larger systems.

Here we focused on providing `R` codes to solve problems like visualization of results over maps. Additionally we wanted that everyone be able to change them in order to answer other questions related to real world problems.

What can we do more? The answer is up to you.