Hands-on network analysis in R

Johan W. Joubert

This document deals with practical examples of building a network graph from activity chain data (as an edge list). This forms part of the *Network analysis* module within the *Sustainable Urban Freight* online course. It is assumed that you have a basic comprehension of R, the language and environment for statistical computing and graphics. Also, in this document we will use some additional R libraries that you may first have to install.

Let's read in the given file and see what we're dealing with.

```r
tours <- read.csv(file="./tours.csv")
str(tours)
```

```
## 'data.frame':    47008 obs. of  4 variables:
##  $ tour: int  1 1 1 1 1 1 2 2 2 2 ...
##  $ trip: int  1 2 3 4 5 6 1 2 3 4 ...
##  $ from: int  830 531 586 282 688 33 718 41 849 436 ...
##  $ to  : int  531 586 282 688 33 830 41 849 436 129 ...
```

```r
head(tours, n=10)
```

```
##    tour trip from  to
## 1     1    1  830 531
## 2     1    2  531 586
## 3     1    3  586 282
## 4     1    4  282 688
## 5     1    5  688  33
## 6     1    6   33 830
## 7     2    1  718  41
## 8     2    2   41 849
## 9     2    3  849 436
## 10    2    4  436 129
```

Each record represents a direct trip made by a vehicle from once facility (`from`) to another (`to`). The direct trips are combined into *tours*, which you can probably also call the *round trip*. We refer to it as the vehicle's *activity chain*. In the first tour we see six trips. The vehicle first travelled from facility 830 to facility 531, then to facility 586, then 282, 688 and 33, and eventually returning to the same starting facility 830. We can also represent the activity chain more graphically as

$$\boxed{830} \rightarrow \boxed{531} \rightarrow \boxed{586} \rightarrow \boxed{282} \rightarrow \boxed{688} \rightarrow \boxed{33} \rightarrow \boxed{830}$$

Edge list

Remember now that each record (row) in the data is a trip. But there may be multiple trips between the same two facilities. Take for example the direct trips from facility 830 to 531.

```r
tours[tours$from=="830" & tours$to=="531", ]
```

We see that there were 324 instances, and it so happens that it was always the first trip in an activity chain. Instead of adding each of these trips as an edge to our graph, we rather want to simply *weigh* the edge by counting how many times a specific trip, or node-pair, occurred. To do that there are a few ways, and we will use the `plyr` library (Wickham et al., 2017).

```
library(plyr)
el <- count(tours, vars=c("from", "to"))
head(el)
```

```
##   from  to freq
## 1    3  38    1
## 2    3  66    1
## 3    3 369    5
## 4    3 576    8
## 5    3 813    7
## 6    3 855    1
```

We refer to the `el` data set as an *edge list* as every row (record) represents an edge in the network. That is, each facility will be represented as a node, and a direct trip between two consecutive nodes is represented by an edge in our graph.

Construction a network graph

In this example we are going to use the `igraph` library in R to build a network graph (Csardi and Nepusz, 2006). For more information about this library you are welcome to visit their website at `http://igraph.org/r/`.

```
library(igraph)
```

```
##
## Attaching package: 'igraph'

## The following objects are masked from 'package:stats':
##
##     decompose, spectrum

## The following object is masked from 'package:base':
##
##     union
```

You can access more detailed help, and documentation for the library by executing the general help function using `?`.

```
?igraph
```

To create the graph we only need to use the edge list information. That is, columns 1 and 2 of our `el` data set.

```
edgeList <- as.matrix(el[, 1:2])
g <- graph_from_edgelist(el=edgeList, directed=TRUE)
summary(g)
```

```
## IGRAPH 94a2136 D--- 873 8722 --
```

In the summary we see that we're dealing with a directed graph (`D`) that has 873 nodes, each representing a facility in our case, and 8722 edges, each representing a direct trip. Once we have our graph the fun really starts. What you do with your graph, however, depends on the type of questions that you want answered.

To start off with, let's just consider the total degree of each node. This can be calculated very efficiently with the following command.

```
degree(graph=g)
```

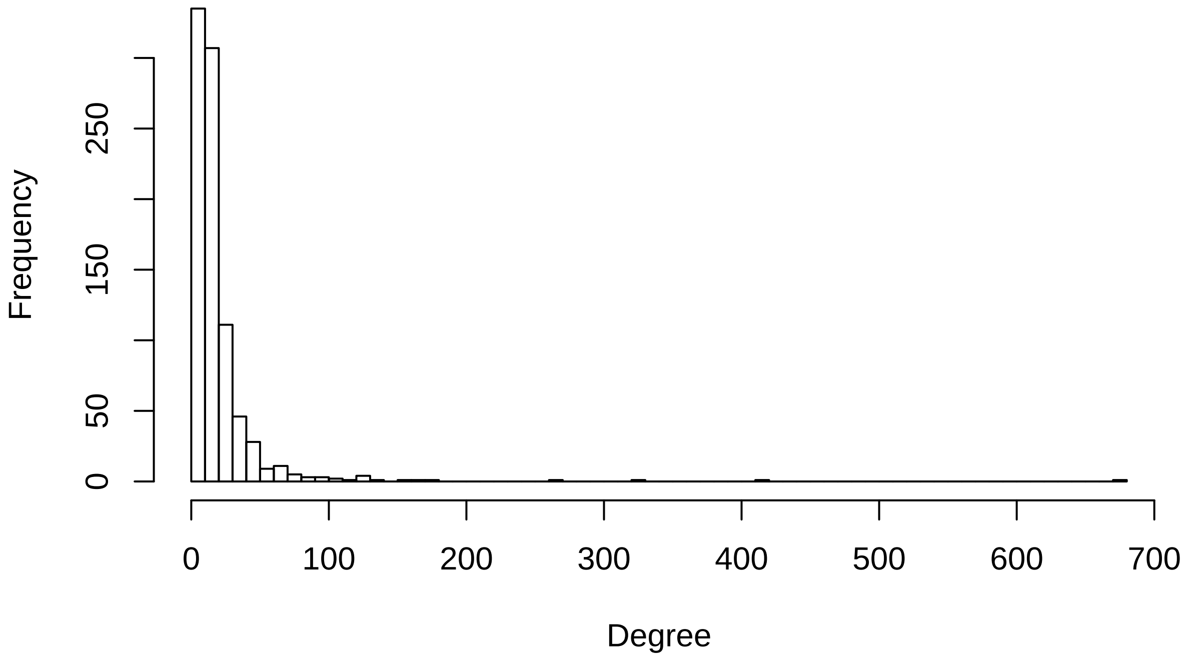
There should be quite a few values, right? To be precise, 873 of them: one for each node. Let's look at a more concise view.

```
summary(degree(g))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    0.00    8.00   13.00   19.98   22.00  679.00
```

To look at the degree distribution more visually, let's plot it as a histogram.

```
hist(degree(g), breaks=50, main="", xlab="Degree")
```



Before we proceed we need to just pause here. Why are there nodes with zero degree values? The answer lies in how `igraph` interprets our facility names, which are numeric integers, `int`, in this case. The first record in our edge list is already `830`. If the node reference is numeric, then `igraph` uses that as the vertex index, so it will already create nodes 1-830, even though nodes 1 and 2 may not even appear in our edge list. And that is why we see nodes 1 and 2 (and many other nodes) having a zero degree.

If, on the other side, we convert the numeric facility references to characters, then `igraph` will assign its own sequential vertex index, and add an attribute `name` to each vertex, and it will use our facility number as the vertex name. So let's retrace our step building the graph and using the facility numbers as a name instead of a vertex index.

```
tours$from <- as.character(tours$from)
tours$to <- as.character(tours$to)
el <- count(tours, vars=c("from", "to"))
edgeList <- as.matrix(el[,1:2])

g <- graph_from_edgelist(el=edgeList, directed=TRUE)
summary(g)
```

```
## IGRAPH 0939ad0 DN-- 822 8722 --
## + attr: name (v/c)
```

Now we see that there are only 822 nodes. You will also see that it is a *directed* (`D`) and *named* graph (`N`). Let's have a look at the first few entries of the edge list that we used to create this graph.

```r
head(el)
```

```
##   from  to freq
## 1   10 205    1
## 2   10 236    3
## 3   10 282    1
## 4   10 306    5
## 5   10  43    1
## 6   10 517    1
```

The `igraph` package can work with weighted networks, but it assumes the weight of a link is the ***cost*** associated with a link, and not the ***strength***. So when the library calculates the *shortest path* through the graph between two nodes, it will do so by minimising ***cost***. Since a high strength is *good* in our case, i.e. makes two facilities *closer* to one another, we have to invert the weights.

We normalise the weights of the edges, the `freq` column, by the average weight of the network. As a consequence, Opsahl et al. (2010) argue that we can have a weight metric that is more comparable across different network. So in the next command we do the normalisation and inversion in one step.

```r
el$weight <- mean(el$freq)/el$freq
head(el)
```

```
##   from  to freq   weight
## 1   10 205    1 5.389590
## 2   10 236    3 1.796530
## 3   10 282    1 5.389590
## 4   10 306    5 1.077918
## 5   10  43    1 5.389590
## 6   10 517    1 5.389590
```

Next we assign the calculated (and normalised) weights to each edge in the network.

```r
E(g)$weight <- el$weight
summary(g)
```

```
## IGRAPH 0939ad0 DNW- 822 8722 --
## + attr: name (v/c), weight (e/n)
```

And now we see that the graph is characterised by '`DNW`' indicating that it is *directed* (`D`), *named* (`N`) and *weighted* (`W`).

Visualising a network

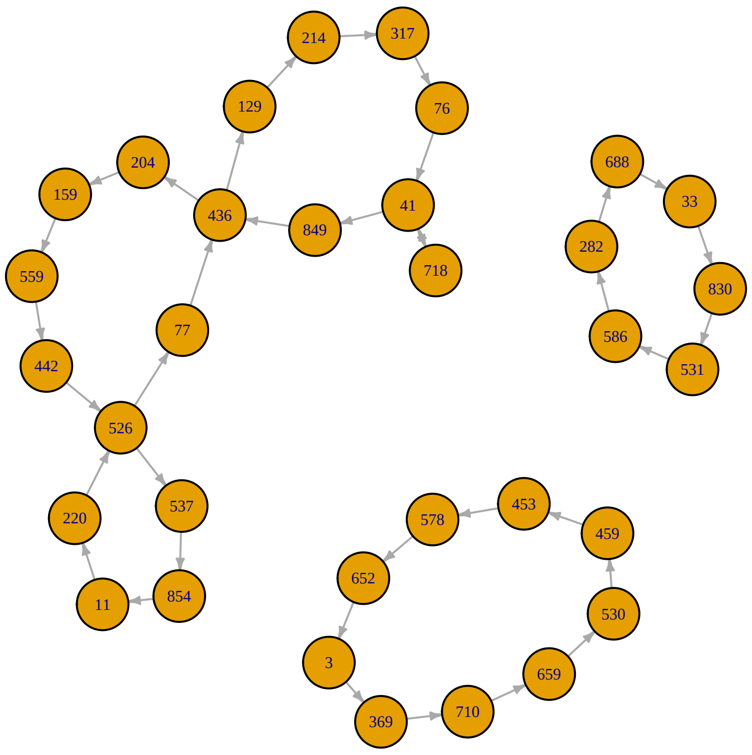
We can try and plot the actual graph, but visualising 822 nodes and 8722 edges will be quite messy. Instead, let's just visualise a small subset. To do that, let's build a graph using only the first 5 tours.

```r
tours.small <- tours[tours$tour <= 5, ]
el.small <- count(tours.small, vars=c("from", "to"))
edgeList.small <- as.matrix(el.small[, 1:2])
g.small <- graph_from_edgelist(el=edgeList.small, directed=TRUE)
E(g.small)$weight <- el.small$freq
summary(g.small)
```

```
## IGRAPH 5dea73d DNW- 33 36 --
## + attr: name (v/c), weight (e/n)
```

And now we can plot it.

```r
plot.igraph(g.small, vertex.label.cex=0.5, edge.arrow.size=0.3)
```



If you want to further prettify your graph, like sizing the vertex labels, or changing the colours, making the edge thickness a function of the edge weight, et cetera, have a look at the plotting help functions.

```r
?plot.igraph
?igraph.plotting
```

If we want to inquire about a specific graph element we use the `V(g)` and `E(g)` calls for vertices and edges, respectively, with the single argument `g` assuming that the graph to which these elements belong is called `g`. Let's look at the first node, or vertex.

```r
V(g)[1]
```

```
## + 1/822 vertex, named, from 0939ad0:
## [1] 10
```

This is indeed the origin node of the first edge we created when building the graph from the edge list. Note, not our initial *tours*, but the edge list. The `igraph` library assigned it its own internal, sequential index of `1`. To make our lives easier, let us create a data frame where we keep track of both the vertex index, and our facility name associated with it. Then we can also add our later calculations to the data frame.

```r
df <- data.frame(id=1:length(V(g)), name=V(g)$name)
```

What if we do not know the internal index, but want to find it, yet only have the vertex name? Try finding the vertex index for facility 830.

```r
which(V(g)$name == "830")
```

```
## [1] 27
```

Or simply using our data frame.

```r
df[df$name=="830", "id"]
```

```
## [1] 27
```

Should you ever need the internal vertex id, you now know how and where to get it.

Network analyses

There is a vast amount of analyses that one can do with a graph. I found the article by Bellamy and Basole (2013) quite useful. They reviewed a large number of literature and group the metrics on three property levels: *node* (vertex), *link* (edge), and *network*. You are welcome to explore these in more detail, but we will stick to some basics in this document.

Node level

We can calculate the vertex metrics for specific graph elements. For example, if we want to know what node 362's degree distribution is, we can calculate it as follows.

```r
degree(g, v="362")
```

```
## 362
## 171
```

Vertex 362's total degree is 171, and we can further distinguish between its in-degree and its out-degree.

```r
degree(g, v="362", mode="in")
```

```
## 362
##  46
```

```r
degree(g, v="362", mode="out")
```

```
## 362
## 125
```

In this data set all activity chains are closed loops, but each trip is not represented as an edge. Instead, trips between the same facilities are aggregated to give each edge a weight. So the in- and out-degrees will not necessarily be the same.

Let us add all the degree metrics to our data frame.

```r
df$degree <- degree(graph=g)
df$inDegree <- degree(graph=g, mode="in")
df$outDegree <- degree(graph=g, mode="out")
```

NExt we calculate node 830's betweenness centrality, another node-level metric.

```r
betweenness(graph=g, v="830", directed=TRUE)
```

```
##   830
## 20020
```

You can, of course, also calculate the betweenness for all nodes in a single command. Let us do that and assign the values to our vertex data frame.

```r
df$bc <- betweenness(graph=g)
```

We can do the same for eigen centrality.

```r
df$ec <- eigen_centrality(graph=g, directed=TRUE)$vector
```

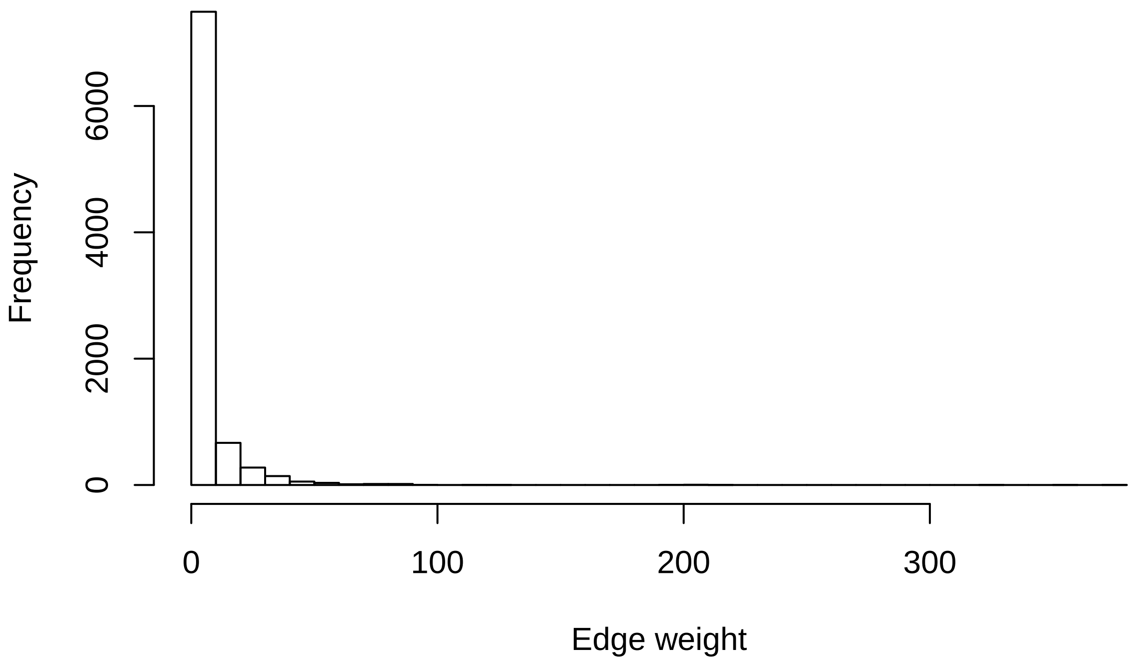
```
## Warning in eigen_centrality(graph = g, directed = TRUE): At centrality.c:
## 362 :Weighted directed graph in eigenvector centrality
```

You'll note that when we calculate the eigen centrality, we actually get a whole list of results from R. In this case we only want to add the actual calculated centrality values that are stored in the `vector` attribute of the result.

Link level

Earlier when we created the network graph we already noted that we are dealing with a weighted graph. We can dig deeper into this by, for example, looking into the weight distribution.

```r
hist(el$freq, breaks=50, main="", xlab="Edge weight")
```



Note that we're using the actual link ***strength*** here, and not the normalised (and inverted) weights as assigned to the `igraph` network. Again, we observe a long, right-tailed distribution.

Network level

Although there are a number of network level properties one can investigate, we will only look at two here. First we look at *centralisation* that measures the extent that one facility, in our case, is more *central* than others.

Yes, we can get the entire metric distribution of all vertices, but we may want to report it as a single metric for the whole graph. And to measure this we need to specify what node-level centrality metric we wish to use. Say we want to use the vertices' degree.

```r
degree.centralisation <- centr_degree(graph=g, mode="total", normalized=FALSE)
str(degree.centralisation)
```

```
## List of 3
##  $ res             : num [1:822] 11 43 8 324 10 11 18 124 13 121 ...
##  $ centralization : num 540694
##  $ theoretical_max: num 1348082
```

The calculation gives you a list with three values: the individual node centrality values used; the graph centrality; and the maximum theoretical graph centralisation score for a graph with the given number of vertices and using the same parameters.

A second metric is *density*. This is the ratio of the number of edges and the number of possible edges. The number of *possible* edges represents the scenario the every vertex is connected to every other vertex in the network.

```
edge_density(graph=g)
```

```
## [1] 0.01292412
```

So we see that only 1.29% of all possible edges exist in our network.

Estimating a power law

The degree distribution of many (real-world) graphs are often best estimated with a power law. Clauset et al. (2009) argued that this is only partially true. Instead, they found that the degree distribution only follows the power law beyong a given degree threshold, often referred to as $x_{\min}$. We can test this in R using the `poweRLaw` library (Gillespie, 2015).

```
library(poweRlaw)
m <- displ$new(df$degree)
e <- estimate_xmin(m)
m$setXmin(e)
print(e)
```

```
## $gof
## [1] 0.02135936
##
## $xmin
## [1] 17
##
## $pars
## [1] 2.743818
##
## $ntail
## [1] 345
##
## $distance
## [1] "ks"
##
## attr(,"class")
## [1] "estimate_xmin"
```

We can perform a bootstrapping hypothesis test to determine if this suggested power law distribution is indeed plausible.

```
bootstrap.p <- bootstrap_p(m, no_of_sims=10, threads=4)
```

```
## Expected total run time for 10 sims, using 4 threads is 1.04 seconds.
```

```
bootstrap.p
```

```
## $p
## [1] 1
##
## $gof
## [1] 0.02135936
##
## $bootstraps
##           gof xmin     pars ntail
## 1  0.02888621   17 2.530634   328
## 2  0.02363963   22 2.595963   203
## 3  0.03424908   16 2.787435   380
## 4  0.04167391   17 2.742615   364
## 5  0.02296263   17 2.647807   343
## 6  0.03196517   20 2.758820   272
## 7  0.02496680   17 2.763518   345
## 8  0.02629975   22 2.937889   235
## 9  0.02536858   17 2.628598   345
## 10 0.02640790   18 2.781275   297
##
## $sim_time
## [1] 2.212256
##
## $seed
## NULL
##
## $package_version
## [1] '0.70.1'
##
## $distance
## [1] "ks"
##
## attr(,"class")
## [1] "bs_p_xmin"
```

The results of ten bootstraps returned a p-value of 1, suggesting the degree distribution *does* follow a power law as $p > 0.1$. Accepting the hypothesis we can estimate the scale parameter, α, as follows.

```
mean(bootstrap.p$bootstraps$pars)
```

```
## [1] 2.717455
```

Recall that the theoretical expression for the power law function is $p(x) \propto x^{-\alpha}$. The lower threshold, x_{min}, beyond which we can accept the hypothesis that the degree distribution is scale free, is estimated as follows.

```
mean(bootstrap.p$bootstraps$xmin)
```

```
## [1] 18.3
```

References

Bellamy, M. A. and Basole, R. C. (2013). Network analysis of supply chain systems: A systematic review and future research. *Systems Engineering*, 16(2):235–249.

Clauset, A., Shalizi, C. R., and Newman, M. E. (2009). Power-law distributions in empirical data. *SIAM review*, 51(4):661–703.

Csardi, G. and Nepusz, T. (2006). The *igraph* software package for complex network research. *InterJournal*, Complex Systems:1695. Available online at http://igraph.org. (Accessed 20 May 2015).

Gillespie, C. S. (2015). Fitting heavy tailed distributions: The poweRlaw package. *Journal of Statistical Software*, 64(2):1–16.

Opsahl, T., Agneessens, F., and Skvoretz, J. (2010). Node centrality in weighted networks: Generalizing degree and shortest paths. *Social Networks*, 32(3):245–251.

Wickham, H., Francois, R., Henry, L., and Mueller, K. (2017). *dplyr: A Grammar of Data Manipulation*. R package version 0.7.4.