SDS First Assignment

Authors: Giovanni Giunta & Adrienn Timea Aszalos

The assigment is divided in two parts.

The first part consists in implementing a random max-cut algorithm and evaluating the average cut-size over M simulations for a specific small graph and then for bigger graph.

The second part consists in simulating the creation of several random networks, generated by a process that respects two main assumptions:

1- The networks we create have to be the result of a “growth” process that continuously increases each network’s size “N” (in random networks the number “N” of nodes is, instead, fixed).

2- When creating new nodes, we need to make sure they are more likely to link themselves to the more connected nodes (“preferential attachment” process).

We then need to conduct several analysis on such networks.

PART A

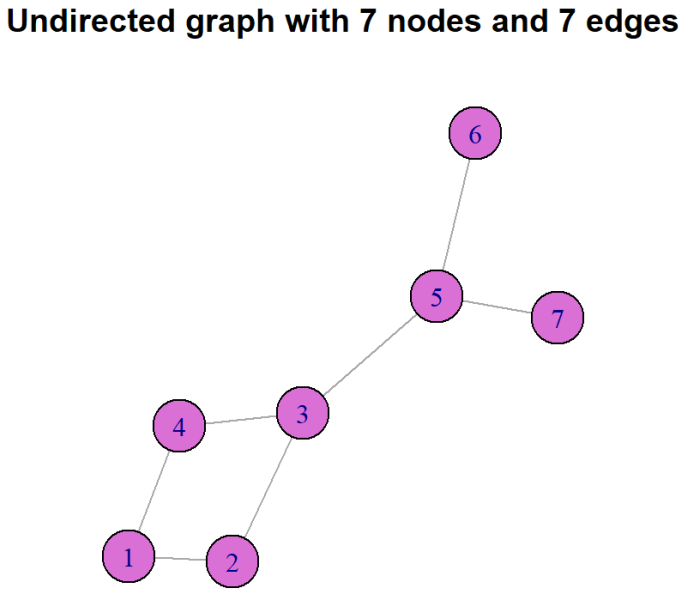
Implementing a randomized max-cut algorithm applied on a specific small graph and evaluating the average cut-size over M simulations

Loading packages

```
suppressPackageStartupMessages(library(igraph))
suppressPackageStartupMessages(library(ggraph))
suppressPackageStartupMessages(library(sdpt3r))
suppressPackageStartupMessages(library(purrr))
suppressPackageStartupMessages(library(poweRlaw))
```

Building a specific graph

```
# Defining the nodes of the graph
graph_nodes <- data.frame(node=c(1,2,3,4,5,6,7))

# Defining the edges of the graph
graph_edges <- data.frame(from=c(1,1,2,4,3,5,5), to=c(2,4,3,3,5,6,7))

# Defining the graph
g <- igraph::graph_from_data_frame(graph_edges, directed=FALSE, vertices=graph_nodes)

# Plotting the graph
plot(g, vertex.size=25, vertex.color="orchid", main="Undirected graph with 7 nodes and 7 edges")
```



Implementing the randomized max-cut algorithm using the previous graph and running it one time

```
# Calculating the starting time of the random max-cut algorithm over the small graph with 7 nodes and 7 edges
startTime <- Sys.time()

# Pre-allocating a vector proportional to the number of the nodes of the graph
vecOfOutcomes <- rep(NA, nrow(graph_nodes))

# Inserting the outcomes, simulated from a coin toss with binomial distribution, into a vector
for (i in 1:nrow(graph_nodes)){
    outcome <- rbinom(1, 1, 0.5)
    vecOfOutcomes[i] <- outcome
}

# Getting the size of set U (U has to be created in the next lines)
count <- 0
for (i in 1:length(vecOfOutcomes)){
    if (vecOfOutcomes[i] == 1){
        count <- count + 1
    }
}

# Pre-allocating and creating the U set randomly, based on the previuos binomial simulation
U <- rep(NA, count)
indexOfU <- 0
for (i in 1:length(vecOfOutcomes)){
    if (vecOfOutcomes[i] == 1){
        indexOfU <- indexOfU + 1
        U[indexOfU] <- graph_nodes[i,1]
    }
}

# Check if the edges are inside set U (XOR conditioning) and get the cut
cut <- 0
for (row in 1:nrow(graph_edges)) {
    first <- graph_edges[row, "from"]
    second  <- graph_edges[row, "to"]
    if (xor(first %in% U,second %in% U)){
        cut <- cut + 1
    }
}
```

The elements of random set U are: 2, 3, 5, 7.

The size-cut obtained from the randomized max-cut algorithm is equal to 3.

Getting the true OPT(G) or at very least a good approximation to OPT(G). G is our previous graph with 7 nodes and 7 edges

```
# Converting the previous graph into a symmetric adjacency matrix
matrix <- as.matrix(as_adjacency_matrix(g,type = c("both")))

# Applying the maxcut algorithm of the package sdpt3r
OPT <- round(maxcut(matrix)$pobj)
OPT <- abs(OPT)
```

The true size-cut obtained from the max-cut algorithm of the package sdpt3r is equal to 7.

Running the randomized max-cut algorithm a large number M of times over the previous small graph

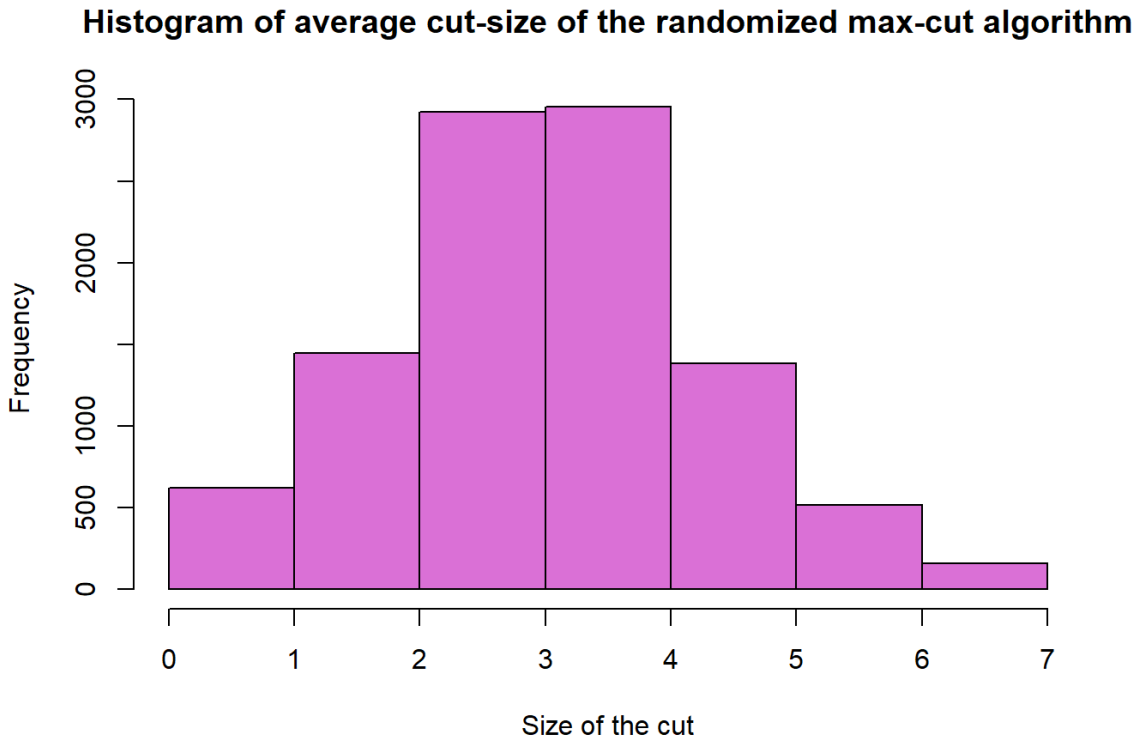
```
# Choosing M equal to 10 000
M = 10000

# Copying and pasting the random algorithm implemented previously and simulating it M times
cutValues <- rep(NA, M)

# Run the randomized algorithm M times
for (j in 1:M){

    # Inserting the outcomes, simulated from a coin toss with binomial distribution, into the vector
    for (i in 1:nrow(graph_nodes)){
        outcome <- rbinom(1, 1, 0.5)
        vecOfOutcomes[i] <- outcome
    }

    # Getting the size of set U (to be created)
    count <- 0
    for (i in 1:length(vecOfOutcomes)){
        if (vecOfOutcomes[i] == 1){
            count <- count + 1
        }
    }

    # Pre-allocating and creating the U set randomly, based on the previuos binomial simulation
    U <- rep(NA, count)
    indexOfU <- 0
    for (i in 1:length(vecOfOutcomes)){
        if (vecOfOutcomes[i] == 1){
            indexOfU <- indexOfU + 1
            U[indexOfU] <- graph_nodes[i,1]
        }
    }

    # Check if the edges are inside set U (XOR conditioning) and get the cut
    cut <- 0
    for (row in 1:nrow(graph_edges)) {
        first <- graph_edges[row, "from"]
        second  <- graph_edges[row, "to"]
        if (xor(first %in% U,second %in% U)){
            cut <- cut + 1
        }
    }

    # Storing each cut-size in a vector
    cutValues[j] <- cut
}

# Calculating the ending time of the random max-cut algorithm over the small graph with 7 nodes and 7 edges,
 simulated M times
endTime <- Sys.time()

# Calculating the running time of the random max-cut algorithm simulated M times over the small graph with 7
 nodes and 7 edges
deltaTime <- endTime-startTime
```

The random max-cut algorithm simulated M+1 times (we consider the first single simulation we did initially and then the M simulations) over the small graph with 7 nodes and 7 edges takes 2.261 seconds.

We will skip showing the all the size-cut values obtained from the previous M simulations, since the list is large, and proceed with a statistical evaluation.

A short statistical analysis over the cut-sized obtained from the M simulations of the randomized max-cut algorithm

```
# Calculating the average size-cut
meanCut <- mean(cutValues)

# Calculating the maximum size-cut
maxCut <- max(cutValues)

# Calculating the minimum size-cut
minCut <- min(cutValues)

# Calculating the standard deviation of the size-cut
stanDev <- sd(cutValues)

# Calculating the coefficient of variation
coeffVariation <- round(stanDev/meanCut,2)

# Drawing the histogram of the average cut-size values of the randomized max-cut algorithm
hist(cutValues, main = paste("Histogram of average cut-size of the randomized max-cut algorithm"), xlab="Size
of the cut", col="orchid", breaks=nrow(graph_edges))
```



The average cut-size over the M simulations is upper bounded by the optimal cut value: 3.5031 => 3.5.

The maximum cut-size obtained from the M simulations is equal to 7.

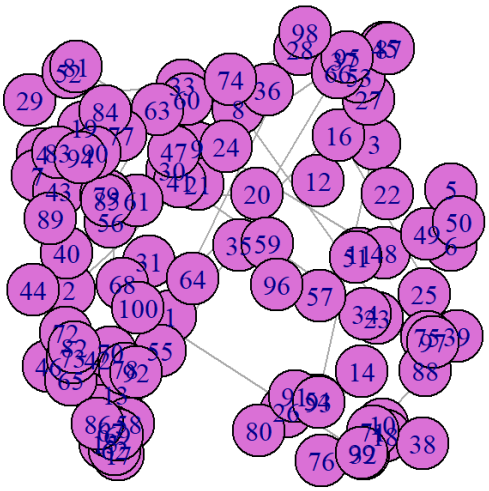
The minimum cut-size obtained from the M simulations is equal to 0.

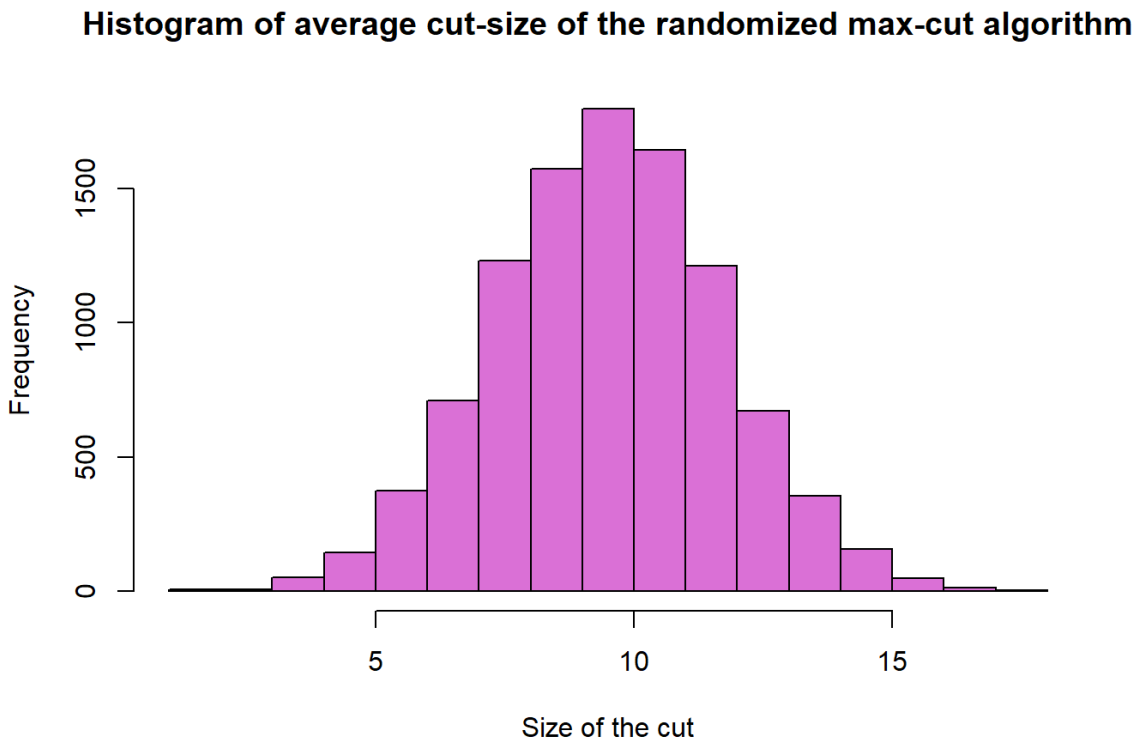
The coefficient of variation is equal to 0.38.

The standard deviation of the cut-size obtained from the M simulations is equal to 1.34.

Changing the graph size to see if there is an impact on the performance of the randomized max-cut algorithm

1. We are applying the same randomized max-cut algorithm on a bigger graph with 100 nodes and 20 edges and doing M simulations to check the impact on the performance of the algorithm.





The average cut-size over the M simulations is upper bounded by the optimal cut value: 9.9897 => 10.

The maximum cut-size obtained from the M simulations is equal to 18.

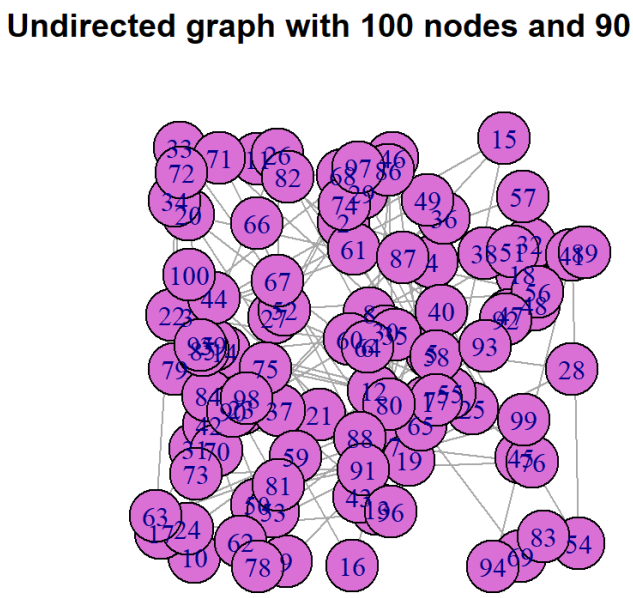
The minimum cut-size obtained from the M simulations is equal to 1.

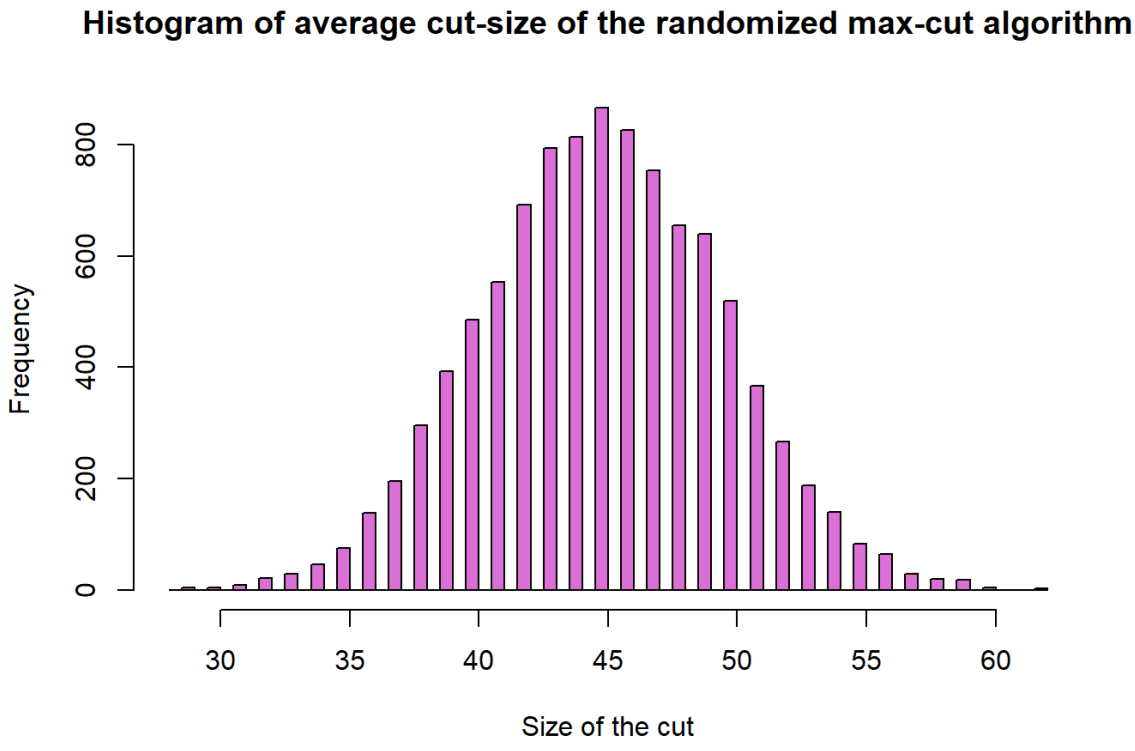
The coefficient of variation is equal to 0.22.

The standard deviation of the cut-size obtained from the M simulations is equal to 2.23.

The random max-cut algorithm simulated M times over the graph with 100 nodes and 20 edges takes 10.24 seconds.

2. We are applying the randomized max-cut algorithm on a similar graph respect to the previous. The graph has 100 nodes, but we are incrementing the size of the edges to 90 and applying M simulations to check the impact on the performance of the algorithm.





The average cut-size over the M simulations is upper bounded by the optimal cut value: 45.0296 => 44.

The maximum cut-size obtained from the M simulations is equal to 62.

The minimum cut-size obtained from the M simulations is equal to 28.

The coefficient of variation is equal to 0.1.

The standard deviation of the cut-size obtained from the M simulations is equal to 4.71.

The random max-cut algorithm simulated M times over the graph with 100 nodes and 20 edges takes 23.676 seconds.

Final conclusions over the performance of the random max-cut algorithm

- From the three histograms we deduce that incrementing the number of simulations of the randomized max-cut algorithm and the number of the edges of the graph the distribution of the average size-cut becomes symmetric. This symmetry is also due to the Binomial Distribution we use in the simulation of the coin toss when choosing randomly the size of the subset U.
- From the three values of the standard deviation of the cut-size obtained from the M simulations we notice that the variability of the size-cuts is medium/low.
- The larger the graphs the smaller is the coefficient of variation (which is a number between 0 and 1). When getting close to 0 the random max-cut algorithm becomes more precise and near in getting the optimal value of the size-cut.
- The bigger the size of the graph, especially with incrementing the number of the edges, the running time of the randomized max-cut algorithm increases

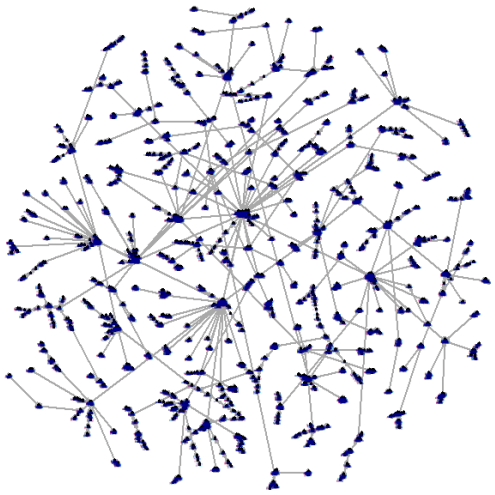
PART B

Simulation of a Preferential Attachment Process over a Randomly Generated Network

- Write a program in R to simulate the preferential attachment process, starting with 4 pages linked together as a directed cycle on 4 vertices, adding pages each with one outlink until there are 1 million pages **(EDIT : as agreed, 100.000 nodes are enough)**, and using $\gamma = 0.5$ as the probability a link is to a page chosen uniformly at random and $1 - \gamma = 0.5$ as the probability a link is copied from existing links.

```
get_random_integer = function(value) {
  # This function picks randomly (uniformly) one of the pages that have been added so
  # far to our network simulation.
  # Since every page is labelled as an integer going from 1 to the current iteration
  # value, we simply generate a random number belonging to such interval. To avoid
  # the very small chance that the link will point to the page itself, we simply
  # consider "max=value" rather than "max=value + 1".

  result = floor(runif(1, min=1, max=value))

  return (result)
}

get_random_link = function(m, value) {
  # In this case we select randomly (uniformly) a single link out of the vector "m".
  # In "m" we have inserted every link SO FAR created (until the current iteration
  # "i", which produced the edge labelled as "value", with i == value), therefore
  # we have a much higher chance to select from "m" a link to a page that has a
  # high number of links attached to it. Since the the current "value" is not in
  # the vector "m", the fact that the generated link will be an "outlink" is guaranteed.

  links = to_vector(m)
  maxVal = length(links) + 1
  result = floor(runif(1, min=1, max=maxVal))

  return (links[result])
}

to_vector = function(listVar) {
  # This is a simple function that converts any list to a vector.

  return (unlist(listVar, use.names=FALSE))
}

set_initial_edges = function() {
  # This is a container for the commands to generate the starting graph
  # containing 4 edges connected to each other as a directed cycle on 4 vertices.

  loop = rep(1:4, each = 2)
  vertices = c(loop[-1], loop[1])

  return (vertices)
}

create_graph_data = function(start, end, gamma) {
  # This is the function that generates a random edge for each newly created vertex.
  # Each vertex is labelled as an integer going from "start" to "end", and can be
  # considered as an additional page added iteratively to our simulation model. Our
  # goal is to associate to each page a single link (edge) to a different page
  # (vertex). Of course, the link has to point to a different page, not to itself.
  # We proceed as follows: we create two different lists, "l" and "m". In "l" we
  # append, at every iteration, the newly created edge AND a randomly generated
  # link pointing to a different page (outlink), while in "m" we append only the
  # randomly generated link.
  # This link will be either:
  #
  # 1- Created so that it picks randomly one of the pages SO FAR existing in the graph
  #    (added up to the current iteration), with probability gamma.
  #
  # 2- Copied among the elements of the list "m", containing all the links existing
  #    SO FAR (added to the graph up to the current iteration), with probability
  #    1 - gamma. Since every existing link is in "m", duplications included, it is way
  #    more probable to copy one of the links pointing to a popular page than to copy any
  #    of those few links pointing to a rarely referenced page.
  #
```

```
  # At the end of the iteration, we are interested in the list "l" containing the sequence
  # of all the edges, going from vertex "i" to vertex "x", for every "i" and "x" belonging
  # to "l". Eventually, we convert "l" to a vector and return the result.

  l = list()
  m = list(1,2,3,4)

  for (i in seq(start,end)) {
    p = rbernoulli(1, p = gamma)

    if (p) {
      x = get_random_integer(i)

    } else {
      x = get_random_link(m, i)
    }

    l = append(list(i,x), l)
    m = append(list(x), m)
  }

  data = to_vector(l)

  return (data)
}

generate_graph = function(end) {
  gamma = 0.5   # the probability of choosing a link at random among all the pages
  initial_vertices = 4   # the number of vertices of the initial graph
  initial_edges = set_initial_edges()   # the edges of the initial graph
  start = initial_vertices + 1    # the starting value of the interval
  increment = end - start + 1   # the number of vertices to add to the initial graph
  data = create_graph_data(start, end, gamma)   # creates an edge for each additional vertex

  grph <- make_empty_graph() %>%          # initialize an empty graph
    add_vertices(initial_vertices) %>%    # add the first 4 vertices
    add_edges(initial_edges) %>%          # add the first 4 edges
    add_vertices(increment) %>%           # add to the initial graph the remaining vertices
    add_edges(data)                       # add to the initial graph the remaining edges

  return (grph)
}

a = generate_graph(10000)

plot(
     a,
     edge.width=1,
     edge.arrow.size=0.1,
     vertex.size=1,
     vertex.label.cex=0.1
     )

title(
  "Randomly generated network with a Preferential Attachment Process \n and 10.000 nodes",
  line = -19
  )
```

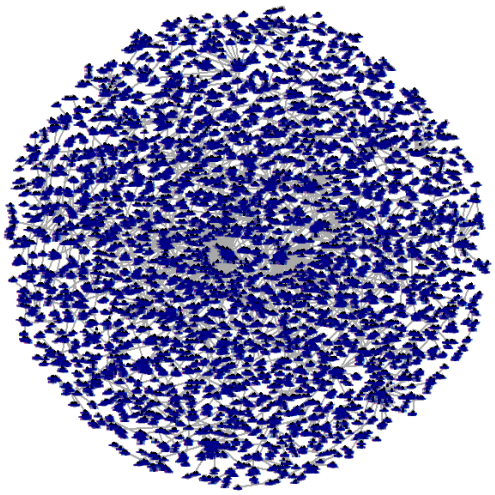


Randomly generated network with a Preferential Attachment Process and 10.000 nodes

In the representation above we can see how the preferential attachment process has favoured the creation of several hubs, that are those nodes with high degrees. The more a hub “grows” with connections during the network creation process, the more we can expect it to continue growing at the expense of nodes with lower degrees. In particular, it is visible how these hubs tend to form huge “clusters” in different parts of the graph.

Let’s now plot another Random network, this time with 100.000 nodes (not a million, but still enough):

```
b = generate_graph(100000)

plot(
     b,
     edge.width=1,
     edge.arrow.size=0.1,
     vertex.size=1,
     vertex.label.cex=0.1
     )

title(
  "Randomly generated network with a Preferential Attachment Process \n and 100.000 nodes",
  line = -19
  )
```



Randomly generated network with a Preferential Attachment Process and 100.000 nodes

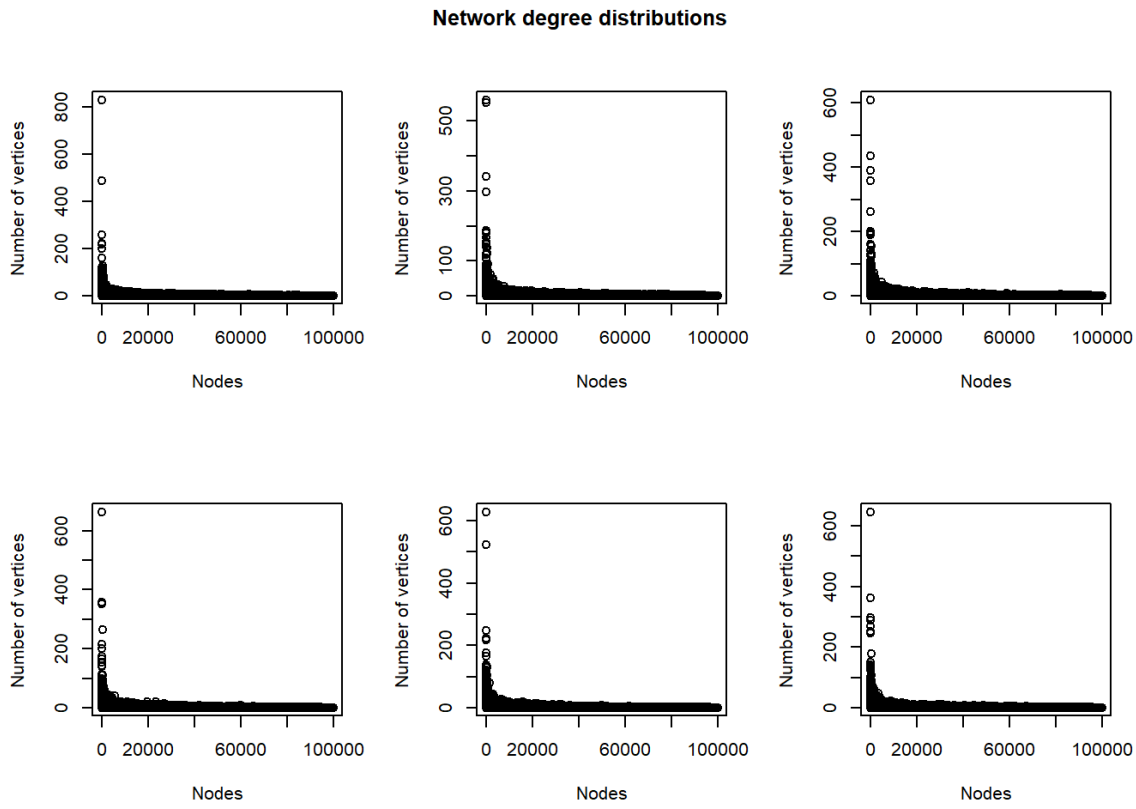
- Simulate a small number M of networks and draw a plot of their empirical degree distribution, showing the number of vertices of each degree on a log–log plot. Also draw a plot showing the complimentary cumulative degree distribution – that is, the number of vertices with degree at least k for every value of k – on a log–log plot.

We now proceed with creating other 5 networks, for a total, counting the one plotted above, of M = 6, all with the same node size of 100.000.

```
c = generate_graph(100000)
d = generate_graph(100000)
e = generate_graph(100000)
f = generate_graph(100000)
g = generate_graph(100000)

graph_list = list(b,c,d,e,f,g)
```

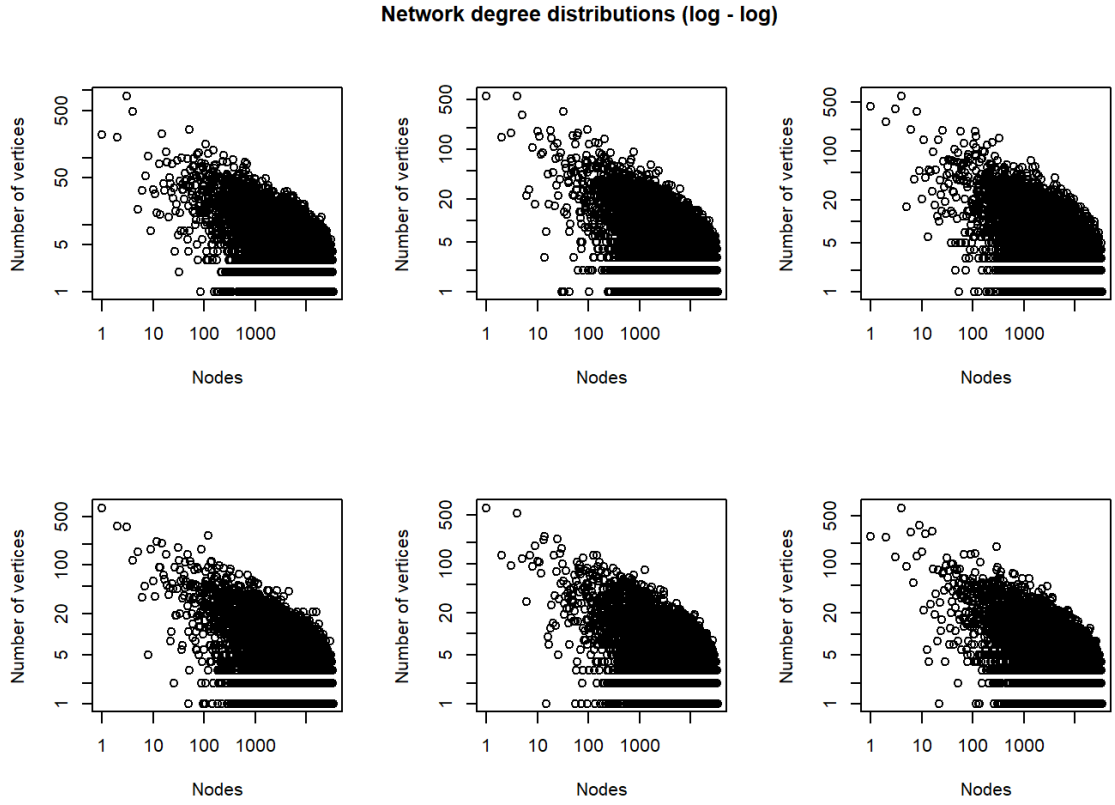
Let's now plot the number of vertices of each degree. We choose the mode="in", while computing the degrees, because we want to consider only the vertices each node is receiving (in-degree). In other words, what we care to know for the purpose of our analysis is how "popular" each page is if compared to the others.

```
options(scipen = 999)

par(mfrow=c(2,3))

for (grph in graph_list) {
  # we use the {igraph} built-in method to count the degree of each node.
  degrees = degree(grph, mode="in")

  plot(degrees, xlab="Nodes", ylab="Number of vertices")
}

title('Network degree distributions', line=-1, outer=TRUE)
```



Below, the network degree distributions are presented in log-log plots:

```
par(mfrow=c(2,3))

for (grph in graph_list) {
  # we use the {igraph} built-in method to count the degree of each node.
  degrees = degree(grph, mode="in")
  filteredDegrees = degrees[degrees > 0]

  plot(filteredDegrees, log="xy", xlab="Nodes", ylab="Number of vertices")
}

title('Network degree distributions (log - log)', line=-1, outer=TRUE)
```



The network degree distributions presented above clearly show how very few nodes have accumulated, during the generation process, lots and lots of vertices, while the vast majority of them got only a few, some barely one. In general, we can see how the chosen approach for generating random networks is working as expected, providing us with data that is consistent with the main assumptions of this assignment:

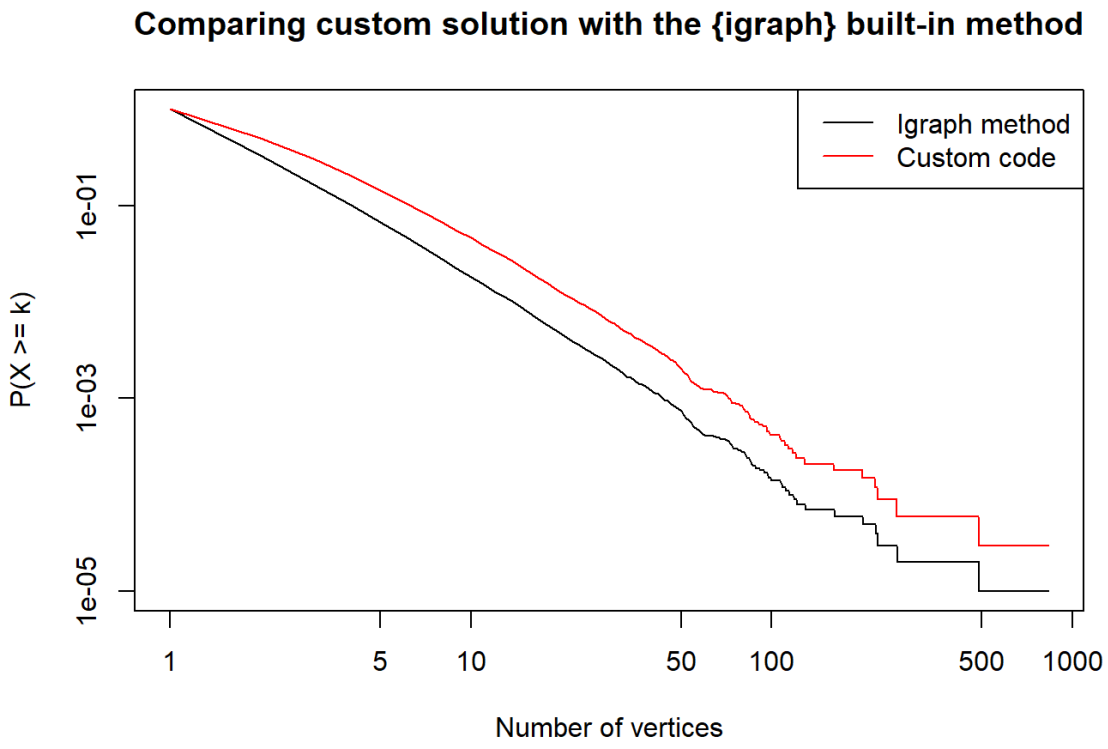
1- Our data is the result of a growth process, in other words, of a process developed upon a starting point (our initial 4 pages linked together as a directed cycle on 4 vertices), and grown randomly iteration after iteration following a discrete time Markov chain.

2- While growing in size, the development of new vertices in each Network has been influenced by the probability of copying one of the existing vertices, which leaded to the creation of several hubs, around which nodes with lesser vertices gravitate around.

It is fair to say that these two assumptions have been respected.

Now let's also plot the complementary cumulative degree distribution on a log-log plot. To do so, we can either use the {igraph} built-in method "degree_distribution()", or use some custom code. As shown below, it appears that there are some discrepancies between the two solutions. We prefer to trust the expertise of the authors of the {igraph} package, and choose to follow their method.

```
selected_grph = graph_list[[1]] # We pick one of the graphs in the list

ccdd_dist = function(deg) {
  dg1 = list();

  for (i in 1:max(deg)) {
    dg1[i] = length(deg[deg >= i]) / length(deg);
  }

  return (dg1)
}

degrees = degree(selected_grph, mode="in")
filteredDegrees = degrees[degrees > 0]

dg1 = ccdd_dist(filteredDegrees)

dg2 = degree_distribution(selected_grph, cumulative=TRUE, mode="in")
dg2 = dg2[dg2 > 0]

options(scipen = 0)

plot(dg2, log="xy", type="l", xlab="Number of vertices", ylab="P(X >= k)")
lines(1:max(filteredDegrees), dg1, col="red")
title("Comparing custom solution with the {igraph} built-in method")
legend("topright",lty=c(1,1),col=c("black", "red"),legend=c("Igraph method","Custom code"))
```



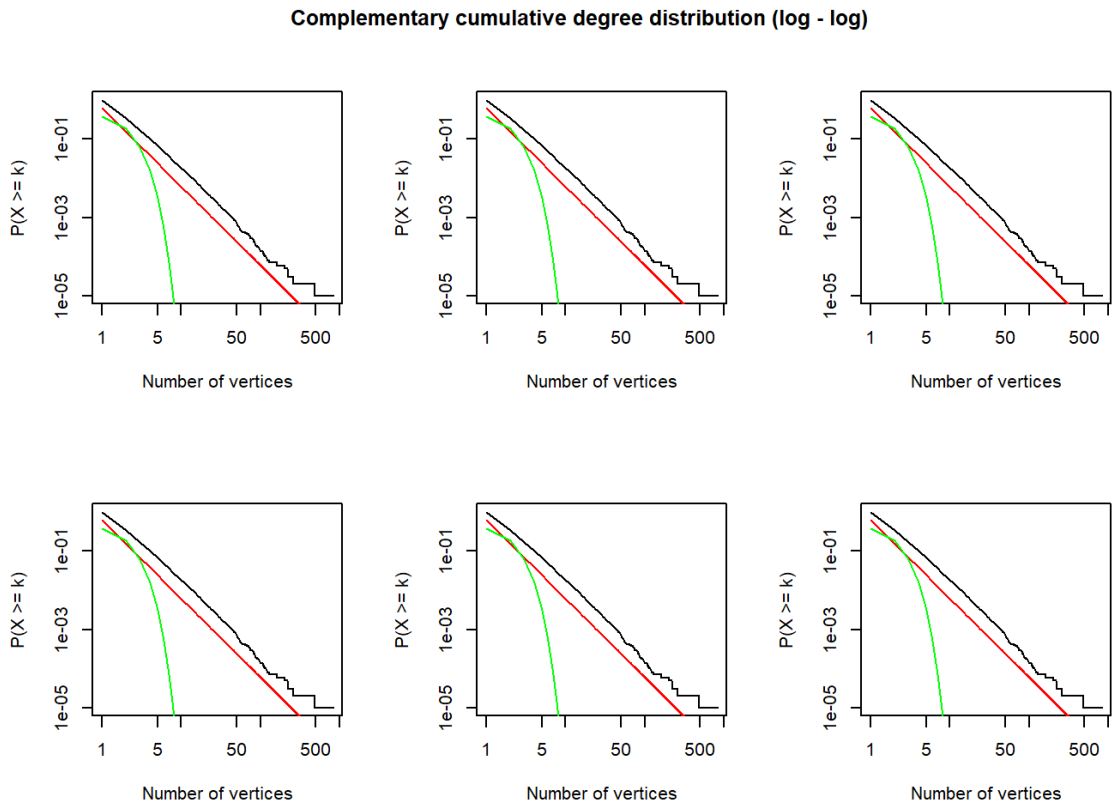
We can now calculate the CCDDs for each one of our random networks, and compare them with:

1- The distribution of a discrete Power Law with alpha = 2 (red)

2- The distribution of a discrete Poisson with lambda = 1 (green)

Even though the processes examined are discrete, we will plot them as lines for visual simplicity.

```
xmin=1
alpha=2
x=xmin:100000

par(mfrow=c(2,3))

for (grph in graph_list) {
  degrees = degree(grph, mode="in")
  filteredDegrees = degrees[degrees > 0]

  degrees = degree_distribution(selected_grph, cumulative=TRUE, mode="in")

  plot(degrees, log="xy", type="l", xlab="Number of vertices", ylab="P(X >= k)")
  lines(x, dpldis(x, xmin, alpha), col="red")
  lines(x, dpois(x, lambda=1), col="green")
}

title('Complementary cumulative degree distribution (log - log)', line=-1, outer=TRUE)
```



We can see how these empirical distributions we obtained are much closer to a Power Law then a Poisson. In general, as stated in the assignment, we can say that the asymptotical behavior of our distributions are indeed very close to a straight line, where the slopeness is given by the true value of alpha. Hence, it looks like our empirical distribution is a very good approximation of a Power Law.

Let's now see if this also proves to be true analytically. We are going to make use of the fit_power_law method of {igraph} over one of our sample networks:

```
selected_grph = graph_list[[1]] # We pick one of the graphs in the list, in this case the first
degrees = degree(selected_grph, mode="in")

fit_power_law(degrees)
```

```
## $continuous
## [1] FALSE
##
## $alpha
## [1] 2.905904
##
## $xmin
## [1] 13
##
## $logLik
## [1] -3352.783
##
## $KS.stat
## [1] 0.01307237
##
## $KS.p
## [1] 0.9960197
```

Behind the scenes, the fit_power_law function has made all the necessary calculations to fit the provided vector named “degrees” (a vector containing the empirical degrees of all the nodes from a selected Network) to a power law distribution.

From the output, we can see that the function correctly recognized our data to be discrete, and accordingly tried to fit a discrete Power Law (continuous: FALSE). The parameter “alpha”, which is the exponent of the fitted discrete Power Law, is calculated according to an optimal estimated value for “xmin”, that is, the minimum empirical value used to fit the Power Law distribution (only the data larger than “xmin” are used from the given vector). Such optimal value is the one for which the p-value of a Kolmogorov-Smirnov test between the fitted distribution and the given empirical set of data is the largest (the p-value is indicated as KS.p). In other words, we want to minimise the distance between our data and the CDF of the fitted model.

Now, the Kolmogorov-Smirnov test produced by fit_power_law has returned a very low test statistic score (KS.stat) and a quite high p-value (Ks.p). High p-values and small scores are both indicators of a good fit of our empirical distribution to a Power Law. More specifically, the closer is the p-value to 1, the more the model (in this case the Power Law) is a plausible fit to our data.

[Igraph fit_power_law][1] [1]: https://igraph.org/r/doc/fit_power_law.html (https://igraph.org/r/doc/fit_power_law.html)

For further confirmation, we can use the procedures and tools provided by the {poweRlaw} package. Again, let’s estimate the values of “xmin” and “alpha”, this time with {poweRlaw}:

```
# in this case, we use filteredDegrees, meaning degrees > 0, as the check_discrete_data method of
# {poweRlaw} only works with strictly positive integers.
m1 = displ$new(filteredDegrees)

# we estimate the value of "xmin" and set it to our function
m1$setXmin(estimate_xmin(m1))

xmin = m1$xmin
alpha = m1$pars

print(xmin)
```

```
## [1] 8
```

```
print(alpha)
```

```
## [1] 2.708917
```

We can see that the values for “xmin” and “alpha” returned by {poweRlaw} are equal to the ones returned by {igraph}. Eventually, we estimate the p-value of the Kolmogorov-Smirnov test by means of the bootstrap_p built-in method of {poweRlaw}, for a given amount of simulations and a given amount of threads.

If the p-value is higher than 0.1, we can accept the hypothesis (or rather, not rule it out, as seen in *[Clauset et al.,Power-Law Distributions in Empirical Data, Section 4.2][1]* [1]: http://tuvalu.santafe.edu/~aaronc/courses/5352/readings/Clauset_Shalizi_Newman_09_PowerlawDistributionsInEmpiricalData.pdf (http://tuvalu.santafe.edu/~aaronc/courses/5352/readings/Clauset_Shalizi_Newman_09_PowerlawDistributionsInEmpiricalData.pdf)), assuming that the number of observations in the model we picked, that is, the first element of the “graph_list”", are compatible with such assumption.)

```
n_cores = parallel::detectCores()   # on my local machine the CPU has 12 cores
no_of_sims = 1000

bs = bootstrap_p(m1, no_of_sims=no_of_sims, threads=n_cores)
```

```
## Expected total run time for 1000 sims, using 12 threads is 42.5 seconds.
```

```
print(bs$p)
```

```
## [1] 0.031
```

```
print(bs$p > 0.1)
```

```
## [1] FALSE
```

- Dig a bit more visually and numerically into some of the networks you generated using the R tools metioned in (1.)

An interesting study we can conduct, is looking at the level of global clusterization of our networks (remember, we mentioned clusters before). We might want, in other words, to measure the probability that the adjacent vertices of a vertex are connected. These are sometimes also called clustering coefficients.

```
selected_grph = graph_list[[1]]   # We pick one of the graphs in the list

transitivity(selected_grph)       # global over the whole graph
```

```
## [1] 0
```

```
transitivity(d, type="average")   # same as the previous, but calculated by averaging the local
```

```
## [1] 0
```

```
                                  # coefficients of each vertex
```

The reason for the result being zero, is given by the fact that the global clustering coefficient is based on triplets of nodes that are connected by two or three undirected ties. Now, since in our model each node can have “n” possible in-vertices, but only 1 out-vertex, the result is obviously zero.

Another 3 important things we could observe in our networks are:

1- The vertex/edge connectivity (or simply connectivity), that is, the minimum number of elements (edges or vertices, depending on the study we are interested in) that we have to remove so that the remaining graph is divided into isolated subgraphs.

```
edge_connectivity(selected_grph)    # Connectivity measure for the edges
```

```
## [1] 0
```

```
vertex_connectivity(selected_grph)  # Connectivity measure for the vertices
```

```
## [1] 0
```

Also in this case, our results are equal to zero. When both edge and vertex connectivity are equal to zero, then we can say that the graph is **disconnected**. This means that **not** every pair of edges OR vertices in our graph are connected to each other.

2- The centrality scores (betweenness centrality measures), for both vertices and edges. They measure the extent to which a vertex or an edge lies on paths between other vertices/edges. Elements with high betweenness may have considerable influence within a network by virtue of their control over information passing between others. They are also the ones whose removal from the network will most disrupt communications between other vertices because they lie on the largest number of paths taken by messages.

[Betweenness Centrality][1] [1]: https://www.sci.unich.it/~francesc/teaching/network/betweeness.html (https://www.sci.unich.it/~francesc/teaching/network/betweeness.html))

```
head(betweenness(selected_grph, directed = TRUE),10)   # Vertices betweenness, first 10 values
```

```
##  [1] 184779 102127 143743 169339    520   5156  35288  12496    108  13504
```

```
head(edge_betweenness(selected_grph, directed = TRUE),10)   # Edges betweenness, first 10 values
```

```
##  [1] 184782 102130 143746 169342     12      6      7     11     13     10
```

3- The skewness of the degree distribution. In most real networks, the degree distribution is highly asymmetric (or skewed). This is because most of the nodes have low degrees, while a small (but significant) fraction of nodes (called hubs) have an extraordinarily high degree. Since the probability of hubs, although low, is significant, the degree distribution (when plotted as a function of the degrees k) shows a long tail, a much fatter one than the tail of the Gaussian or exponential model. By measuring the skewness of the degree distribution we can check how symmetric it is. If it proves to be right skewed, than the distribution contains many low values and relatively few high values. Vice versa if it is left skewed. In our case, we want to check that the distribution is **right skewed**. We now proceed with calculating the level of skewness, and whether it is left or right skewed. As a general rule, if the mean is higher than the median, we can say that the distribution is right skewed.

[Power laws and scale-free networks][1] [1]: http://users.dimi.uniud.it/~massimo.franceschet/ns/plugandplay/scale-free/scale-free.html#1 (http://users.dimi.uniud.it/~massimo.franceschet/ns/plugandplay/scale-free/scale-free.html#1))

```
skewness = function(x) {
  mean( (x - mean(x))^3 ) / sd(x)^3
}

z = degree(selected_grph, mode="in")

print(skewness(z))
```

```
## [1] 77.87788
```

```
print(mean(z) > median(z))
```

```
## [1] TRUE
```