AC562 - Complex Systems

Report of the practical work of complex system

Alexandre Tisserand

15/01/2021

TP1

1) Introduction

The aim of this TP is to rise in competence with graph representations, metrics and the software R.

2) Representations of graph and first metrics

1. Matrix <-> List

A graph may be represented by its adjacency matrix or its adjacency list.
Below is a function that gets the adjacency list from the adjacency matrix :

```
fromMatriceToList <- function(mat)
{
  k <- nrow(mat)
  lst <- list()
  for(i in 1:k)
  {
    lst[[i]] <- numeric()
  }
  for (i in 1:k)
  {
    for(j in i:k)
    {
      if( mat[i,j] == 1)
      {
        lst[[i]] <- c(lst[[i]],j)
        lst[[j]] <- c(lst[[j]],i)
      }
    }
  }
  lst
}
```

And now the opposite a function that gets the adjacency matrix form the adjacency list :

```
fromListToMatrice <- function(lst)
{
k <- length(lst)
mat <- vector(mode = "numeric",k*k)
dim(mat) <- c(k,k)
for( i in 1:k )
{
    l <- length(lst[[i]])
    for(j in 1:l)
    {
    mat[i,lst[[i]][j]] <- 1
    }
}
mat
}
```

2. List and distribution of degrees.

Here is a function which return the list of degrees from the adjacency matrix or adjacency list :

```
listOfDegrees <- function(input)
{
  if( is.vector(input) == TRUE )
  {
    k <- length(input)
    lst <- list()
    for( i in 1:k)
    {
      lst[[i]] <- length(input[[i]])
    }
  }
  else if( is.matrix(input) == TRUE )
  {
    templst <- fromMatriceToList(input)
    k <- length(templst)
    lst <- list()
    for( i in 1:k)
    {
      lst[[i]] <- length(templst[[i]])
    }
  } else
  {
    print("Wrong input of function listOfDegrees")
  }
  lst
}
```

Here is a function which return the degrees distribution from the adjacency list (a vérifier et posser une question quand a la définition de la question)

```
degreeDistribution <- function(input)
{
  lstdeg <- listOfDegrees(input)
  print(lstdeg[[which.max(lstdeg)]])

  lstdistrib <- list()
  lstdistrib <- replicate(lstdeg[[which.max(lstdeg)]],0)
  k <- length(lstdeg)
  for(i in 1:k)
  {
    lstdistrib[[lstdeg[[i]]]] <- lstdistrib[[lstdeg[[i]]]] + 1
  }
  print(lstdistrib)
  print("Endfunction")
}
```

3. Clustering coefficient

Faire ici un rappel de ce qu'est le clustering coef.. (maybe donner la def cc(i) = ...)

Here is a function that retruns the list of cluster coefficients from the adjacency matrix.

```r
clustering_coef <- function(input)
{
  len <- nrow(input)
  lst <- list()
  for(i in 1:len)
  {
    output <- 0
    for(k in 1:len)
    {
      for(p in 1 :len )
      {
        output <- output + input[i,k]*input[k,p]*input[p,i]
      }
    }
    lst[[i]] <- output
  }
  lst
}
```

3. Breadth-first search algorithm and applications

1/

Reminder : Breadth-first search algorithm give the distance between a given node to all the other node belonging to the same component.

The first implementation we use is "navie" and use the folloing principles :

Let n = number of nodes.
Let m = number of edges.
Let i = studied node.
Let s = desitination node.
Let d = number of round.

1/ Creation of an array Ds of n intergers to store the distances d(i,s).
2/Initialisation of Ds with Ds(s)=0 and Ds(i) = -1, **quelque soit** i=!s. d=0.
3/ Find all nodes with distance d. If there is no, then stop.
4/ Find the neighbors of these nodes, assign those neighbors which don't have a distance yet, with the distance d + 1.
5/ Set d = d + 1 and go to 3.

Bellow is a R implementation:

```r
breadth_first <- function(input_mat,node)
{
  k <- nrow(input_mat)
  outvect <- vector(mode = "numeric",k)
  for( i in 1:k )
  {
    if( i == node )
    {
      outvect[i] = 0
    }
    else{
      outvect[i] = -1
    }
  }
  end <- FALSE
  d <- 0
  while( end == FALSE  )
  {
    positivecondition <- list()
    for(i in 1:k)
    {
      if( outvect[i]== d)
      {
        positivecondition[length(positivecondition)+1] = i
      }
    }
    if(length(positivecondition) == 0 )
    {
      end <- TRUE
    }else{
      for(i in 1:length(positivecondition))
      {
        for(j in 1:k)
        {
          if( (areneighbours(input_mat,positivecondition[[i]],j) == TRUE) &&
(outvect[j] == -1) )
          {
            outvect[j] <- d + 1
          }
        }
      }
    }
    d <- d + 1
  }
  outvect
}
```

With the details of the areneighbours function bellow.

```r
areneighbours <- function(input,i,j)
{
  if( is.vector(input) == TRUE )
  {
    matinput <- fromListToMatrice(input)
    if(matinput[i,j] == 1 )
    {
      output <- TRUE
    }else
    {
      output <- FALSE
    }
  }
  else if( is.matrix(input) == TRUE )
  {
    if(input[i,j] == 1 )
    {
      output <- TRUE
    }else
    {
      output <- FALSE
    }
  }
  output
}
```

The way we implement this could be better executed (notably the with the areneighbours) but with this algorithm, for a typical network, complexity is O(m + n log n).

A better implementation could be done with using a queue.
Here is a implementation with the stack algorithm :

```
breadth_first_stack <- function(input_mat,node)
{
  aplist <- fromMatriceToList(input_mat)
  stack <- vector(mode = "numeric")
  stack[1] <- node
  read <- 1
  write <- 2
  k <- nrow(input_mat)
  ds <- vector(mode = "numeric",k)
  for( i in 1:k )
  {
    if( i == node )
    {
      ds[i] = 0
    }
    else{
      ds[i] = -1
    }
  }
  while(read != write)
  {
    for(i in 1:length(aplist[[stack[read]]]))
    {
      if(ds[ aplist[[stack[read]]][i] ] == -1 )
      {
        ds[ aplist[[stack[read]]][i] ] <- ds[ stack[read]] + 1
        stack[write] <- aplist[[stack[read]]][i]
        write <- write +1
      }
    }
    read <- read + 1
  }
  ds
}
```

With the stack approch the complexity is samller. O(m + n)

For this tow apporch we have the result of a unique node. To have the complete matrice of distance as required in Q1, we repeat the operation for all nodes and aggregate the outputs. This is the purpose of the mat_D_breadth_first_stack function using the stack algorithm.

```r
mat_D_breadth_first_stack <- function(input_mat)
{
  k <- nrow(input_mat)
  outputmat <- vector(mode = "numeric")

  for(i in 1:k)
  {
    outputmat <- c(outputmat,breadth_first_stack(input_mat,i));
  }
  dim(outputmat) <- c(k,k)
  outputmat
}
```

2/
For Question N°2:
Diameter is a record of the largest distance observed in a component.
Once the Breadth-first search executed, finding the diameter for a given node is straightforward.

```r
diammeter <- function(input_mat,node)
{
  k <- nrow(input_mat)
  bfs_node <- breadth_first_stack(input_mat,node);
  diameter <- max(bfs_node)
  lst_compenent_memeber <- list()
  for(i in 1:k)
  {
    if(bfs_node[i] > 0 )
    {
      lst_compenent_memeber[[length(lst_compenent_memeber)+1]] <- i
    }
  }
  l <- length(lst_compenent_memeber)
  for(j in 1:l)
  {
    if( max(breadth_first_stack(input_mat,lst_compenent_memeber[[j]])) >
diameter )
    {
      diameter <-
max(breadth_first_stack(input_mat,lst_compenent_memeber[[j]]))
    }
  }
  diameter
}
```

3/
Closeness centrality represente

First we have to underline that if the network is not composed of a single unique component, the result of the closeness centrality must be take with care. It is possible to handle each component individualy, but this can bias the values. Indeed nodes in smaller component may have higher value. This is what the following example do:

```r
closeness_centrality_node <- function( input_mat , node )
{
  dist <- breadth_first_stack_list(input_mat,node)
  n <- length(dist)
  sum <- 0
  for(i in 1:n)
  {
    if( dist[[i]] > 0 ) #Here nodes that are not in the component are not take in account.faire la remarque page 47 (A modfifier ?)
    {
      sum <- sum + dist[[i]]
    }
  }
  output <- length(dist)/(sum)
  output
}
```

We use the very unelegant (but functional) breadth_first_stack_list function described bellow.

```r
breadth_first_stack_list <- function(input_mat,node)
{
  output_vect <- breadth_first_stack(input_mat,node)
  k <- length(output_vect)
  output_list <- list()
  for(i in 1:k)
  {
    if(output_vect[i] >= 0)
    {
      output_list[[length(output_list)+1]] <-  output_vect[i]
    }
  }
  output_list
}
```

The igraph package throw a waring when the components are not all linked togethers.

4/ Implementing the Betweennes centrality was for the hardest task to complete. In fact finding all the shortest path is the blocking point.
I tried to do this job with a recusive function knowing the lenght of every shortest path with a previous breadfirst search.
Unfortunately I did not maneged to make this function work. The following programm visit all the nodes, but returning the target point and aggregate the coresting path is not so easy.

4. With the package Igraph

First you need to download the igraph libray via the R packet mananger:

```r
install.packages("igraph")
```

Then add load the package at the beging of the script

```r
library(igraph)
```

1/
The function undirectedIgraphFromAdjancyMatrix that defines an undirected graph in Igraph from an adjacency matrix.

```r
undirectedIgraphFromAdjancyMatrix <- function(input_mat)
{
  nbnode <- nrow(input_mat)
  graph  <- make_empty_graph(directed = FALSE)
  igraph <- add_vertices(graph,nbnode, color = "red")
  for(i in 1: nbnode)
  {
    for(j in i:nbnode)
    {
      if(input_mat[i,j] == 1)
      {
        igraph <- add_edges(igraph, c(i,j))
      }
    }
  }
  igraph
}
```

The function directedIgraphFromAdjancyMatrix do the same job but with a directed graph.

```r
directedIgraphFromAdjancyMatrix <- function(input_mat)
{
  nbnode <- nrow(input_mat)
  graph  <- make_empty_graph(directed = TRUE)
  igraph <- add_vertices(graph,nbnode, color = "red")
  for(i in 1: nbnode)
  {
    for(j in 1:nbnode)
    {
      if(input_mat[i,j] == 1)
      {
        igraph <- add_edges(igraph, c(i,j))
      }
    }
  }
  igraph
}
```

2/
Using igraph you can have **degree** with

```r
igraphdegrees <- function(igraph)
{
  degree(igraph)
}
```

The **global clustering coefficient** :

latex : C :=number of triangles * 3 / number of connected triples

```r
global_clustering_coefficients <- function(igraph)
{
  transitivity(igraph)
}
```

Local clustering coefficient for a given node.

Reminder of the formula : Latex ! => Ci := number of pairs of neighbors of i that are connected/number of pairs of neighbors of i

```r
local_clustering_coefficient <- function(igraph,node)
{
  transitivity( igraph, type = "local")[node]
}
```

The **normalized closeness centrality** is :

```r
igraph_Closeness_centrality_node <- function(mattestbis,node)
{
  closeness(ig, mode="in",normalized = FALSE)[node]*nrow(mattestbis)
}
```

Betweenness_centrality of a node is :

```r
igraph_betweenness_centrality_node <- function(input_mat,node)
{
  igraph <- undirectedIgraphFromAdjancyMatrix(input_mat)
  betweenness(igraph,normalized = FALSE)[node]
}
```

In the next section we will compare the results of the igraph function and the function we made.

TO BE COMPLETED ....

3/ It's possible the change the color of the node acordingly to there caratecistic.

Color

To determine the color of a node we use a linear repartition of atribute with rgb color.
You can use a color of refernce like the one used in the exemple: esisar's purple color.

```r
inputcolor.R <- 146
inputcolor.G <- 39
inputcolor.B <- 143

findcolor <- function(input_val,low_val, high_val,inputcolor )
{
  range <- high_val - low_val;
  step.R <- floor(inputcolor.R * (input_val-low_val) / range)
  step.G <- floor(inputcolor.G * (input_val-low_val) / range)
  step.B <- floor(inputcolor.B * (input_val-low_val) / range)
  step <- c(step.R/256,step.G/256,step.B/256)
}
```

The we just have to mofify the vertex attribute.

```r
colorgraph <- function(igraph,inputcolor,inputtype)
{
  if(inputtype == "degree" )
  {
    degreevect <- igraphdegrees(igraph)
    for(i in 1:length(degreevect))
    {
      vcolor <- findcolor( degree(igraph)[i] , min(degree(igraph)), max(degree(igraph)) , inputcolor )
      igraph <- set.vertex.attribute(igraph, 'color', i, rgb(vcolor[1],vcolor[2],vcolor[3]))
    }
  }else if( inputtype == "closeness" )
  {
    closnessvect <- closeness(igraph,normalized = FALSE)
    for(i in 1 : length(closnessvect) )
    {
      vcolor <- findcolor( closnessvect[i] , min(closnessvect), max(closnessvect) , inputcolor )
      igraph <- set.vertex.attribute(igraph, 'color', i, rgb(vcolor[1],vcolor[2],vcolor[3]))
    }
  }else if( inputtype == "clustering" )
  {
    local_cluster <- transitivity(igraph, type = "local")
    for(i in 1 : length(local_cluster) )
    {
      if( is.nan(local_cluster[i]) == FALSE )
      {
        # /!\ Points were local clustering can't be etablish are not colored here
        vcolor <- findcolor( local_cluster[i] , min(local_cluster, na.rm = TRUE), max(local_cluster, na.rm = TRUE) , inputcolor )
        igraph <- set.vertex.attribute(igraph, 'color', i, rgb(vcolor[1],vcolor[2],vcolor[3]))
      }
    }
  }else if( inputtype == "betweenness" )
  {
    betw <- betweenness(igraph,normalized = FALSE)
    for(i in 1 : length(betw) )
    {
      if( is.nan(betw[i]) == FALSE )
      {
        vcolor <- findcolor( betw[i] , min(betw, na.rm = TRUE), max(betw, na.rm = TRUE) , inputcolor )
        igraph <- set.vertex.attribute(igraph, 'color', i, rgb(vcolor[1],vcolor[2],vcolor[3]))
      }
    }
  }
  igraph
}
```

Size

The process to modify the node size is almost the same as for color. We have replace the input color by a size coeficient to change the scale of ploting.

```r
findsize <- function(input_val,low_val, high_val, sizeCoef )
{
  range <- high_val - low_val;
  step  <-  ((input_val-low_val) / range)*sizeCoef
}
```

```r
sizegraph <- function(igraph,inputtype,sizeCoef)
{
  if(inputtype == "degree" )
  {
    degreevect <- igraphdegrees(igraph)
    for(i in 1:length(degreevect))
    {
      wsize <- findsize( degree(igraph)[i] , min(degree(igraph)), max(degree(igraph)) , sizeCoef )
      igraph <- set.vertex.attribute(igraph, 'size', i, wsize)
    }
  }else if( inputtype == "closeness" )
  {
    closnessvect <- closeness(igraph,normalized = FALSE)
    for(i in 1 : length(closnessvect) )
    {
      wsize <- findsize( closnessvect[i] , min(closnessvect), max(closnessvect) , sizeCoef )
      igraph <- set.vertex.attribute(igraph, 'size', i, wsize)
    }
  }else if( inputtype == "clustering" )
  {
    local_cluster <- transitivity(igraph, type = "local")
    for(i in 1 : length(local_cluster) )
    {
      if( is.nan(local_cluster[i]) == FALSE )
      {

        wsize <- findsize( local_cluster[i] , min(local_cluster, na.rm = TRUE), max(local_cluster, na.rm = TRUE) , sizeCoef )
        igraph <- set.vertex.attribute(igraph, 'size', i, wsize)
      }else
      {
        # /!\ Points were local clustering can't be etablish we set default size to 1
        igraph <- set.vertex.attribute(igraph, 'size', i, 1)
      }
    }
  }else if( inputtype == "betweenness" )
  {
    betw <- betweenness(igraph,normalized = FALSE)
    for(i in 1 : length(betw) )
    {
      if( is.nan(betw[i]) == FALSE )
      {
        wsize <- findsize( betw[i] , min(betw, na.rm = TRUE), max(betw, na.rm = TRUE) , sizeCoef )
        igraph <- set.vertex.attribute(igraph, 'size', i, wsize)
      }
    }
  }
  igraph
}
```

Those function can be perforemed more efficiantly.

TP2

1) Introduction

TP2 is focused on creation of undirected random graph model. Once the model builded, we uses metrics to extract some of there caracteristiques.

2) Erdos-Renyi model

The Graph Erdos-Renyi G(n, p) = (V,E) constructed from a set V of n vertices.
The edge between 2 vertices i and j exists with probability p.

Below is a propostion of algorthiym :

```
Erdos_Renyi <- function(n,p)
{
  igraph  <- make_empty_graph(directed = FALSE)
  igraph <- add_vertices(igraph,n, color = "red")
  for(i in 1 : n)
  {
    for(j in i : n)
    {
      if( runif(1) < p )
      {
        igraph <- add_edges(igraph, c(i,j))
      }
    }
  }
  igraph
}
```

This implementation give the right output. Hoever due to the 2 for loop performance is quite bad and it can take sevral minutes with large graph.

A way faster implementation is porposed below with approch closer to adjency matrix.
Note the transpose trick used to make the matrix symetric.

```
Erdos_Renyi_optimized <- function(n,p)
{
  nb_tri <- n*(n-1)/2
  mat <- diag(0,nrow = n, ncol = n)
  vect_rand <- runif(nb_tri , min = 0, max = 1) - p
  for(i in 1:nb_tri)
  {
    if(vect_rand[i]>0)
    {
      vect_rand[i] <- 0
```

```r
    }
    else
    {
      vect_rand[i] <- 1
    }
  }
  mat[lower.tri(mat)] <- vect_rand
  mat <- t(mat)
  mat[lower.tri(mat)] <- vect_rand
  igraph <- graph_from_adjacency_matrix(mat,mode = c("undirected"))
  igraph
}
```

Dire quelques mots sur la distribution de type poisons

3) Watts-Strogatz model : Small world network

The Watts–Strogatz model produces graphs with small-world properties.

During the implementation of this algotium the first version created completed the task enficiently but was very slow (multpiple minutes for n=1000) due to multiple for loops. After rewriting the the r code here is implementation that perform well and can produce the required quasi instantly.

```r
Watts_Strogatz_opt <- function(n,p,m)
{
  igraph <- make_ring(n, directed = FALSE)
  ## End step1
  for(j in 1:m)
  {
    for( i in 1: n)
    {
      igraph <- add_edges(igraph, c(i,((i+j)%%n+1)  ))
    }
  }
  ## End step2
  nb_tri <- n*(n-1)/2
  vect_rand <- runif(nb_tri , min = 0, max = 1) - p
  for(i in 1:nb_tri)
  {
    if(vect_rand[i]>0)
    {
      vect_rand[i] <- 1
    }
    else
    {
      vect_rand[i] <- 0
    }
  }
  sp <- matrix( nrow = n, ncol = n)
  sp[lower.tri(sp)] <- vect_rand
  sp <- t(sp)
  sp[lower.tri(sp)] <- vect_rand
  diag(sp) <- 0

  adj_mat <-  as_adjacency_matrix(igraph,type = c("both"))
  res_mat <- adj_mat*sp
  igraph <- graph_from_adjacency_matrix(res_mat,mode = c("undirected"))

  ## End Step 3 (remouving edges)

  act_size <- gsize(igraph)
  aim_size <- n*(2+2*m)/2
  nb_edges_to_add <- aim_size - act_size
  for( i in 1: nb_edges_to_add)
  {
    edge_added <- FALSE
    while(edge_added == FALSE)
    {
      proposal <- floor(runif(2,min = 1, max = n))
      if( (are_adjacent(igraph, proposal[1], proposal[2] ) == FALSE ) && ( proposal[1] != proposal[2] ) )
      {
        igraph <- add_edges(igraph, c( proposal[1], proposal[2] )  )
        edge_added <- TRUE
      }
    }
  }
  igraph
}
```

4) Graph scale free

The following code genrate a scale free graph for any type of k.

```r
nb_init_nodes <- function(nb_init_edges)
{
  n_cal <- (1+sqrt(1+8*nb_init_edges))/2
  if( n_cal == floor(n_cal) )
  {
    return(n_cal)
  }else{
    n_cal <- floor(n_cal)+1
  }
  n_cal
}

scale_free_init <- function(k)
{
  nb_node <- nb_init_nodes(k)
  nb_tri <- nb_node*(nb_node-1)/2
  mat <- matrix( nrow = nb_node, ncol = nb_node)
  diag(mat) <- 0
  vect_init <- c( rep(1, k) , rep(0, (nb_tri-k) ))
  mat[lower.tri(mat)] <- vect_init
  mat <- t(mat)
  mat[lower.tri(mat)] <- vect_init
  igraph <- graph_from_adjacency_matrix(mat,mode = c("undirected"))
  igraph
}

scale_free_degree_range <- function(igraph,random)
{
  random <- runif(1)
  vect_deg <- degree(igraph)
  nbedges <- sum(vect_deg)/2
  sum <- 0
    for(i in 1:length(vect_deg))
    {
        sum <- sum + vect_deg[i]/(nbedges*2)
        if( sum > random)
        {
          return(i)
        }
    }
}

scale_free <- function(n,k,q)
{
  igraph <- scale_free_init(k)
  add <- 0
  nodenb <- k
  while( nodenb < n)
  {
    igraph <- add_vertices(igraph,1)
    for(i in 1:q)
    {
      igraph <- add_edges(igraph,c( (nodenb-1+q),scale_free_degree_range(igraph)))
    }
    nodenb <- nodenb + 1
  }
  igraph
}
```

5) Histograms of the degree distribution.

First we will generate diffrents grap with the previous functions:

- Erdos-Renyi with 1000 nodes and diferent values of the probability p from 0 to 1 with a step of 0.05.

To do this we use the folling code:

```
generate_ER <- function(n)
{
  p <- 0
  lst = list()
  for(i in 1:21)
  {
    lst[[i]] <- degree(Erdos_Renyi_optimized(n,p))
    p <- p + 0.05
  }

  find_xmax_histo <- function(lst_deg)
  {
    max_histo <- 0
    for( i in 2: length(lst_deg)-1)
    {
      if( max(lst_deg[[i]]) > max_histo )
      {
        max_x_histo <- max(lst_deg[[i]])
      }
    }
    max_x_histo
  }

  draw_histos <- function(lst_deg,x_max)
  {
    cat("Max histo = ",x_max)
    max_y_histo <- 0
    #for( i in 1: )
    for( i in 2: length(lst_deg)-1)
    {
      strmain <- c('Histogram of Erdos_Renyi for p=',(i-1)*0.05)
      vect_hist <- hist(lst_deg[[i]],
                        main=strmain,
                        xlim=c(0,x_max),
                        breaks=(x_max+2))
      if(max(vect_hist$counts) > max_y_histo )
      {
        max_y_histo <- max(vect_hist$counts)
      }
    }
    max_y_histo
  }
  xmax <- find_xmax_histo(lst)
  draw_histo(lst,xmax)
}
```

We can observe clearly see the Poisson distribution with the diffent probability.

- Watts-Strogatz of size n = 1000 with p = 0.1 and m = 2.

```
Watts_histo <- function(igraph)
{
  vect_deg <- degree(igraph)
  hsacle <- max(vect_deg)
  hist(degree(ig),
       main="Watts Strogatz Histogram \n ( n = 1000 , k = 2 , p = 0.1 ) ",
       xlim=c(2,hsacle),
       breaks=(hsacle+2),
       col= rgb(146/256,39/256,143/256)
  )
}
```

[PUT ONE IMG]

- A scale free of size n = 1000 with k = 3 and q = 2.

For this scale free we have added a power law in comparaison to verify this law.

```
customhisto <- function(igraph)
{
  vect_deg <- degree(igraph)
  tot <- sum(vect_deg, na.rm = FALSE)
  hsacle <- max(vect_deg)
  pow <- powerlaw(100,hsacle)
  hist(degree(ig),
       xlim=c(2,hsacle),
       breaks=(hsacle+2),
       col= rgb(146/256,39/256,143/256)
       )
  lines(pow[2,]*tot,lwd = 2,
  col = "green")
}
```

Here is the code used to create the power law :

```
powerlaw <- function(hsacle,nbpt)
{
  a <- 2.5
  C <- 1
  step <- hsacle/nbpt

  mat <- vector(mode = "numeric",(2*nbpt))
  dim(mat) <- c(2,nbpt)

  for(i in 1:nbpt)
  {
    mat[1,i] <- i*step
    mat[2,i] <- C*i^(-a)
  }
  mat
}
```

[PUT ONE IMG]

6) Scale free, power law degree distribution

To show more the distribution of degree in scale free graph follow power law of type P(k) ~k^-a.

We used a cumulative distribution visualisation with the fllowing code

[PUT FORMULA if we have time]

```
sortdeg <- function(igraph)
{
  vect_deg <- degree(igraph)
  sorted <- sort.int(vect_deg,decreasing = TRUE)
  sortedind <- sort.int(vect_deg,decreasing = TRUE, index.return = TRUE)

  scale_max <- max(sortedind$x)
  mat <- vector(mode = "numeric",(2*scale_max))
  dim(mat) <- c(2,scale_max)
  for(i in 1: length(sortedind$x))
  {
    mat[2,sortedind$x[i]] <- mat[2,sortedind$x[i]]+1
  }
  for( i in length(mat[2,]):2 )
  {
    mat[2,i-1] <- mat[2,i-1] + mat[2,i]
  }
  normal <- mat[2,1]
  for( i in 1 : length(mat[2,]))
  {
    mat[1,i] <- i
    mat[2,i] <- mat[2,i]/normal
  }
  max_x <- max(sortedind$x)
  print(max_x)
  plot(mat[1,],mat[2,],
       xlab = "Degree k",
       ylab = "Fraction of nodes Pk having degree k or greater",
       xlim = c(1,max_x),
       ylim = c(0.001,1),
       log ="xy"
       )
  # We may have pb with k = 2..
}
```

[PUT ONE IMG]

7) Erdos-Renyi , average length as function of p

To calculate the avreage lenght with the following formula.

[PUT FORMULA]

We use the follwing code :

```
average_length <- function(igraph)
{
  print(vcount(igraph))
  sum <- 0
  for( i in 1: vcount(igraph) )
  {
    bfs_i <- bfs(igraph,i,
        unreachable = FALSE,
        dist = TRUE);
    for(j in i : vcount(igraph) )
    {
      if( i != j)
      {
        #cat("i:",i,"\t","j:",j,"\tdist:",bfs_i$dist[j],"\n");
        #bfs_i$dist[j]
        if( !is.nan( bfs_i$dist[j] ) )
        {
          sum <- sum + bfs_i$dist[j]
        }
      }
    }
  }
  l <- 2*sum/( (vcount(igraph)*( vcount(igraph) -1 ))  )
}
```

The evolution of the average length with p is ploted with the folling code :

```
p_average_length <- function(nb_nodes,nb_sample)
{
  p <- 0
  step <- 1/nb_sample
```

```r
  mat <- vector(mode = "numeric",(2*(nb_sample+1)))
  dim(mat) <- c(2,nb_sample+1)
  for(i in 1: (nb_sample + 1))
  {
    mat[1,i] <- p
    ig <- Erdos_Renyi_optimized(nb_nodes,p)
    mat[2,i] <- average_length(ig)
    p <- p + step
    cat("Progress: ",(i/(nb_sample+1))*100,"%\n")
  }
  strmain <- c("Evolution of average length with p for Erdos-Renyi n=",nb_nodes)
  plot(x = mat[1,],
       y = mat[2,],
       main = strmain,
       type = "b",
       xlab = "p",
       ylab = "Average length",
       )
}
```

[PUT ONE IMG]

8) Erdos-Renyi, clustering coefficient as functions of p

The same pinciple is applyed for evolution of the global clustering coefficient.

```r
p_clustering_coef <- function(nb_nodes,nb_sample)
{
  p <- 0
  step <- 1/nb_sample
  mat <- vector(mode = "numeric",(2*(nb_sample+1)))
  dim(mat) <- c(2,nb_sample+1)
  for(i in 1: (nb_sample + 1))
  {
    mat[1,i] <- p
    ig <- Erdos_Renyi_optimized(nb_nodes,p)
    mat[2,i] <- transitivity(ig)
    p <- p + step
    cat("Progress: ",(i/(nb_sample+1))*100,"%\n")
  }
  strmain <- c("Evolution of the global clustering coefficient with p for Erdos-Renyi n=",nb_nodes)
  plot(x = mat[1,],
       y = mat[2,],
       main = strmain,
       type = "b",
       xlab = "p",
       ylab = "Global clustering coefficient",
  )
}
```

[PUT ONE IMG]

TP3

1) Introduction

In this TP, we will study a voter model. The social interaction will be modelized with an undirected graph. After the creation of a dynamic modilisation, we focus on diffrents scenarios that would allow you to influence the opinion.

2) Voter model

The model described in TP3 is a simple agent-based on an undirected graph. Each node has only one attribute that represente his vote.

First we initialise the network a Bernoulli's law of parameter 0.5 :

```r
initVoteBernoulli <- function(igraph)
{
  for( i in 1: vcount(igraph))
  {
    if( runif(1) < 1/2 )
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=1)
    }
    else
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=0)
    }
  }
  igraph
}
```

Then each node is influence by his neigbors.
Here is fast implementation that work close to adjency matrix :

```r

Nb_neighbors_vect <- function(igraph)
{
  mat <- as_adjacency_matrix(igraph)
  len <- nrow(mat)
  N_vect <- vector(mode = "numeric",len)
  for(i in 1 : len )
  {
    res <- sum(mat[i,])
    if(res > 0)
    {
      N_vect[i] <- sum(mat[i,])
    }else if(res == 0)
    {
      N_vect[i] <- 1
    }
    else{
      print("BUG - Nb_neighbors_vect")
    }
  }
  N_vect
}

probability_vect <- function(igraph,N_vect)
```

```r
{
  adj_mat <-  as_adjacency_matrix(igraph, sparse = FALSE)
  vect_vote <- get.vertex.attribute(igraph, "vote")
  prob_vect <- adj_mat%*%vect_vote
  out <- prob_vect/N_vect
  out
}

getOneVote <- function(igraph)
{
  vc <- vcount(igraph)
  vect <- vector(mode = "numeric",vc)
  for( i in 1 : vc)
  {
    vect[i] <- get.vertex.attribute(igraph, "vote",i)
  }
  vect
}

vote_probability <- function(igraph,noise,N_vect)
{
  p_v <- probability_vect(igraph,N_vect)
  len <- length(p_v)
  f_p_opti <- vector(mode = "numeric",len)
  f_p_opti <- (1-2*noise)*p_v + noise
  f_p_opti
}

vote <- function(igraph,noise,N_vect)
{

  f_p <- vote_probability(igraph,noise,N_vect)
  vc <- vcount(igraph)
  for( i in 1: vc)
  {
    if( runif(1) < f_p[i] )
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=0)
    }
    else
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=1)
    }
  }
  igraph
}
declare_winner <- function(mat)
{
  end_vote_vect <- mat[,ncol(mat)]

  if( sum(end_vote_vect)/nrow(mat) == 0.5  )
  {
    cat("Deuce\n")
  }else if( sum(end_vote_vect)/nrow(mat) < 0.5 )
  {
    cat("Jerry win\n")
  }else if( sum(end_vote_vect)/nrow(mat) > 0.5 )
  {
    cat("Tom win\n")
  }
}

simulation <- function(igraph,noise,time)
{
  N_vect <- Nb_neighbors_vect(ig)

  igraph <- initVoteBernoulli(igraph)
  mat <- vector(mode = "numeric",(vcount(igraph)*time))
  dim(mat) <- c(vcount(igraph),time)
  for(i in 1: time)
  {
    mat[,i] <- getOneVote(igraph)
    igraph <- vote(igraph,noise,N_vect)
    cat(i," over ",time,"\n")
  }
  declare_winner(mat)
  mat
}
```

Then you just have to give the network of you choice to the simulation and you will have complete matrix that repesente the evolution of votes.

Before the developpement of this implementation of a dynamic model I have build an other model witch is way less time-efficient.
Here is the code:

```r
probability_node <- function(igraph, node)
{
  adj_lst_node <- as_adj_list(igraph)[[node]]
  if( length( adj_lst_node ) == 0 )
  {
    return(0)
  }
  sum <- 0
  for(i in 1 : length( adj_lst_node ) )
  {
    sum <- sum + get.vertex.attribute(igraph, "vote",    adj_lst_node[i]
)
  }
  out <- sum/ length( adj_lst_node )
  out
}

probability_vect <- function(igraph)
{
  vc <- vcount(igraph)
  vect <- vector(mode = "numeric",vc)
  sum <- 0
  for( i in 1 : vc )
  {
    sum <- sum + probability_node(igraph,i)
    vect[i] <- probability_node(igraph,i)
  }
  vect
}
```

```r

probability_tot <- function(igraph)
{
  vect <- probability_vect(igraph)
  out <- sum(vect) / vcount(igraph)
}

getOneVote <- function(igraph)
{
  vc <- vcount(igraph)
  vect <- vector(mode = "numeric",vc)
  for( i in 1 : vc)
  {
    vect[i] <- get.vertex.attribute(igraph, "vote",i)
  }
  vect
}

vote_probability <- function(igraph,noise)
{
  p_v <- probability_vect(igraph)
  len <- length(p_v)
  f_p <- vector(mode = "numeric",len)
  for(i in 1 : len  )
  {
    f_p[i] <- (1-2*noise)*p_v[i] + noise
  }
  f_p
}

vote <- function(igraph,noise)
{
  start_time <- Sys.time()
  end_time <- Sys.time()
  f_p <- vote_probability(igraph,noise)
  end_time <- Sys.time()
  exec_time <- (end_time-start_time)
  print(exec_time)
  vc <- vcount(igraph)
  for( i in 1: vc)
  {
    if( runif(1) < f_p[i] )
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=0)
    }
    else
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=1)
    }
  }
  igraph
}

simulation <- function(igraph,noise,time)
{
  igraph <- initVoteBernoulli(igraph)
  mat <- vector(mode = "numeric",(vcount(igraph)*time))
  dim(mat) <- c(vcount(igraph),time)
  for(i in 1: time)
  {
    mat[,i] <- getOneVote(igraph)
    igraph <- vote(igraph,noise)
    cat(i," over ",time,"\n")
  }
  mat
}
```

Notice the diffrence intoduce by working with matrix in comparaison with for loops.

The evolution of the voting rate can be repsented via the output matrix, however this data still need to be simplifyed in order to be more human readable. Here the role of the declare winner function is to inform us about the winner if the vote is done at the end of the simulation.

3) Influenceing scenarios

With a scale free graph of 501 nodes(to avoid Deuce), k = 3, m = 2, and for simulation of 0.01 noise and time = 3000, we will try to influence the vote.
We will work in Jerry's Team.

a) Scenario A

We have the possibiliy convince 10 people to vote for Jerry. We will use the metrics to identify them.

We want to have :

- Node connected to the giant conponent.
- Node with a samll lenght to all the others nodes.
- Node with high closness centrality.

Functions n_length, reacheable_node and how_closness are used to identify nodes with an high influence.

On the other side we have to take in consideration the limitation on the sum of degree of 100. First we use the how_bad_is_degree function. Then the function high_cut also play a role to eliminate the nodes with highest degree with a non linerar function.

All thoses paramter can be tuned by hand with coefficient in order to impouve the ranking.

Then to ensure that we do respect the degree condition we cycle down the top result until the condition is respected.

Here is the code to do the selection of node to influence:

```
selection <- function(igraph)
{
  vect_len        <- n_length(igraph)
  vect_reach      <- reacheable_node(igraph)
  vect_close      <- how_closness(igraph)
  vect_bad_degree <- how_bad_is_degree(igraph)

  coef_len   <- 3
  coef_reach <- 5
  coef_close <- 1.5

  coef_bad_degree <- 1
```

```r
  alpha <- 0.013

  vect_rank_positif <- vect_len*coef_len + vect_reach*coef_reach + vect_close*coef_close

  vect_rank_negatif <- vect_bad_degree*coef_bad_degree
  vect_rank_negatif <- high_cut(vect_rank_negatif,alpha)
  print(vect_rank_negatif)
  print(degree(igraph))

  vect_rank_tot <- vect_rank_positif - vect_rank_negatif

  nb_elect <- 10
  pool <- vector(mode = "numeric",nb_elect)
  ref <- 1
  condition_is_ok <- FALSE
  while(condition_is_ok == FALSE)
  {
    sorted <- sort.int(vect_rank_tot,decreasing = TRUE, index.return = TRUE)
    for(i in ref : (ref+nb_elect) )
    {
      pool[(i-ref)] <- sorted$ix[i]
    }
    vect_deg <- degree(igraph)
    somm <- 0
    #print( vect_deg[pool] )
    for(i in 1:length(pool))
    {

      somm <- somm + vect_deg[pool[i]]
    }
    if(100 > somm)
    {
      cat("Number of rank down to respect condition=",ref," (SUM of degree=",somm,")\n")
      condition_is_ok <- TRUE
    }
    ref <- ref + 1
  }
  pool
}

high_cut <- function(inputvect,alpha)
{
  outputvect <- inputvect + alpha*inputvect*inputvect
}

how_closness <- function(igraph)
{
  tryCatch( vect_close <- closeness(igraph,normalized = TRUE) , warning=function() print("-") )
  if( min(vect_close) == max(vect_close) )
  {
    print("May have pb w: how_closness")
    return( c(rep(0,vcount(igraph)) ))
  }
  n_comp_min <- min(vect_close)
  vect_close <- vect_close-n_comp_min
  n_comp_max <- max(vect_close)
  vect_close <- vect_close*100/n_comp_max
  vect_close
}

how_bad_is_degree <- function(igraph)
{
  vect_deg <- degree(igraph)
  if( min(vect_deg) == max(vect_deg) )
  {
    print("May have pb w: how_bad_is_degree")
    return( c(rep(0,vcount(igraph)) ))
  }
  n_comp_min <- min(vect_deg)
  vect_deg <- vect_deg-n_comp_min
  n_comp_max <- max(vect_deg)
  vect_deg <- (vect_deg*100/n_comp_max)+1
  vect_deg
}



reacheable_node <- function(igraph)
{
  Comp = clusters(igraph)
  vect_reach <- Comp$csize[Comp$membership]
  if( min(vect_reach) == max(vect_reach) )
  {
    #print("One giant Component")
    return( c(rep(0,vcount(igraph)) ))
  }
  n_comp_min <- min(vect_reach)
  vect_reach <- vect_reach-n_comp_min
  n_comp_max <- max(vect_reach)
  vect_reach <- vect_reach*100/n_comp_max
  vect_reach
}


n_length <- function(igraph)
{
  nb_node <- vcount(igraph)
  n_length_vect <- vector(mode = "numeric",nb_node)
  for( i in 1:  nb_node)
  {
    bfs_i <- bfs(igraph,i,
                 unreachable = FALSE,
                 dist = TRUE);
    #print(bfs_i$dist)
    n_length_vect[i] <- sum(bfs_i$dist,na.rm = TRUE)/nb_node
  }
  n_length_max <- max(n_length_vect)

  for( i in 1:  nb_node)
  {
    if( n_length_vect[i] == 0 )
    {
      n_length_vect[i] <- n_length_max
    }
  }
  n_length_min <- min(n_length_vect)


  n_length_vect <- (n_length_vect) - n_length_min
  n_length_max <- max(n_length_vect)
  n_length_vect <- 100 - (n_length_vect*100/n_length_max)
}
```

Then the set node chossen is tested in simulation, with they vote force to 1. To ensure that that the our methods is ok we repeat simulation multiple time and observe the result. Bellow is the code to test the model.

```
initVoteBernoulli <- function(igraph,sway_vector)
{
  for( i in 1: vcount(igraph))
  {
    if( runif(1) < 1/2 )
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=1)
    }
    else
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=0)
    }
  }
  nb_sway <- length(sway_vector)
  for( i in 1 : nb_sway )
  {
    igraph <- set.vertex.attribute(igraph,"vote", sway_vector[i] ,value=0)
  }
  igraph
}

vote <- function(igraph,noise,N_vect,sway_vector)
{

  f_p <- vote_probability(igraph,noise,N_vect)
  vc <- vcount(igraph)
  for( i in 1: vc)
  {
    if( runif(1) < f_p[i] )
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=0)
    }
    else
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=1)
    }
  }
  nb_sway <- length(sway_vector)
  for( i in 1:nb_sway )
  {
    igraph <- set.vertex.attribute(igraph,"vote", sway_vector[i] ,value=0)
  }
```

```r
  igraph
}
declare_winner <- function(mat)
{
  end_vote_vect <- mat[,ncol(mat)]
  out <- vector(mode = "numeric",2)

  if( sum(end_vote_vect)/nrow(mat) == 0.5  )
  {
    cat("Deuce\n")
  }else if( sum(end_vote_vect)/nrow(mat) < 0.5 )
  {
    cat("Jerry win with ",(sum(end_vote_vect)/nrow(mat)),"\n")
    out[1] <- 0
  }else if( sum(end_vote_vect)/nrow(mat) > 0.5 )
  {
    cat("Tom win with ",(sum(end_vote_vect)/nrow(mat)),"\n")
    out[1] <- 1
  }
  out[2] <- (sum(end_vote_vect)/nrow(mat))
  out
}


simulation <- function(igraph,noise,time,sway_vector)
{
  N_vect <- Nb_neighbors_vect(igraph)

  igraph <- initVoteBernoulli(igraph,sway_vector)
  mat <- vector(mode = "numeric",(vcount(igraph)*time))
  dim(mat) <- c(vcount(igraph),time)
  for(i in 1: time)
  {
    mat[,i] <- getOneVote(igraph)
    igraph <- vote(igraph,noise,N_vect,sway_vector)
    #cat(i," over ",time,"\n")
  }
  win <- declare_winner(mat)
}

n <- 501
k <- 3
q <- 2

igraph <- scale_free(n,k,q)
sway_vector <- selection(igraph)

noise <- 0.01
time <- 3000

out <-  0
nb_simu <- 100
winner <- 0
score <- 0
for(i in 1: nb_simu )
{
  out <- simulation(igraph,noise,time,sway_vector)
  winner <- winner + out[1]
  score <- score + out[2]
  cat("Progress:",i*100/nb_simu,"%\n")
}
cat("TOM win in ",winner/nb_simu,"%\n")
cat("Avreage score is",score/nb_simu,"%\n")
```

Unfortunaly the results did not seem to be influenced by our method.

b) Introducting Zealots.

Zealots are people that can't change of opinion. To add them into our model we first define on witch node they are:

```
zelot <- function(igraph)
{
  nb_node <- floor(0.4*vcount(igraph))
  nb_node
  vect_zelot <- vector(mode = "numeric",vcount(igraph))
  for(i in 1: vcount(igraph) )
  {
    if( runif(1)>0.8 )
    {
      vect_zelot[i] <- 1
    }else if( runif(1) < 0.2 )
    {
      vect_zelot[i] <- 0
    }else
    {
      vect_zelot[i] <- -1
    }
  }
  vect_zelot
}

initZelot <- function(igraph,zelot_vector)
{
  for( i in 1: vcount(igraph))
  {
    if( zelot_vector[i] == 1 )
    {
      igraph <- set.vertex.attribute(igraph,"zelot", i ,value=1)
    }
    else if (zelot_vector[i] == 0 )
    {
      igraph <- set.vertex.attribute(igraph,"zelot", i ,value=0)
    }else
    {
      igraph <- set.vertex.attribute(igraph,"zelot", i ,value=-1)
    }
  }
  igraph
}
```

The voting function are adapted:

```
initVoteBernoulli <- function(igraph,sway_vector,zelot_vector)
```

```r
{
  for( i in 1: vcount(igraph))
  {
    if( runif(1) < 1/2 )
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=1)
    }
    else
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=0)
    }
  }
  for( i in 1: vcount(igraph))
  {
    if( zelot_vector[i] == 1 )
    {
      igraph <- set.vertex.attribute(igraph,"zelot", i ,value=1)
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=1)
    }
    else if (zelot_vector[i] == 0 )
    {
      igraph <- set.vertex.attribute(igraph,"zelot", i ,value=0)
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=0)
    }else
    {
      igraph <- set.vertex.attribute(igraph,"zelot", i ,value=-1)
    }
  }
  nb_sway <- length(sway_vector)
  for( i in 1 : nb_sway )
  {
    igraph <- set.vertex.attribute(igraph,"vote", sway_vector[i] ,value=1)
  }
  igraph
}

vote <- function(igraph,noise,N_vect,sway_vector,zelot_vector)
{

  f_p <- vote_probability(igraph,noise,N_vect)
  vc <- vcount(igraph)
  for( i in 1: vc)
  {
    if( runif(1) < f_p[i] )
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=0)
    }
    else
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=1)
    }
  }
  for( i in 1: vcount(igraph))
  {
    if( zelot_vector[i] == 1 )
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=1)
    }
    else if (zelot_vector[i] == 0 )
    {
      igraph <- set.vertex.attribute(igraph,"vote", i ,value=0)
    }
  }

  nb_sway <- length(sway_vector)
  for( i in 1:nb_sway )
  {
    igraph <- set.vertex.attribute(igraph,"vote", sway_vector[i] ,value=1)
  }
  igraph
}

```

Then as for the others parameters we remouve them from the ranking with the following trick:

```r
is_zelot <- function(igraph)
{
  nb <- vcount(igraph)
  vect_zelot <- vector(mode = "numeric",nb)
  for( i  in 1: nb)
  {
    if( get.vertex.attribute(igraph, "zelot",i) == 1 || get.vertex.attribute(igraph, "zelot",i) == 0 )
    {
      vect_zelot[i] <- 99999999
    }else if(get.vertex.attribute(igraph, "zelot",i) == -1 )
    {
      vect_zelot[i] <- 0
    }
  }
  vect_zelot
}

selection <- function(igraph)
{
  vect_len        <- n_length(igraph)
  vect_reach      <- reacheable_node(igraph)
  vect_close      <- how_closness(igraph)
  vect_bad_degree <- how_bad_is_degree(igraph)
  vect_zelot      <- is_zelot(igraph)

  coef_len   <- 3
  coef_reach <- 5
  coef_close <- 1.5

  coef_bad_degree <- 1
  alpha <- 0.013

  vect_rank_positif <- vect_len*coef_len + vect_reach*coef_reach + vect_close*coef_close

  vect_rank_negatif <- vect_bad_degree*coef_bad_degree + vect_zelot
  vect_rank_negatif <- high_cut(vect_rank_negatif,alpha)
  vect_rank_tot <- vect_rank_positif - vect_rank_negatif

  print(vect_rank_tot)

  nb_elect <- 10
  pool <- vector(mode = "numeric",nb_elect)
  ref <- 1
  condition_is_ok <- FALSE
  while(condition_is_ok == FALSE)
  {
    sorted <- sort.int(vect_rank_tot,decreasing = TRUE, index.return = TRUE)
    for(i in ref : (ref+nb_elect) )
    {
      pool[(i-ref)] <- sorted$ix[i]
    }
    vect_deg <- degree(igraph)
    somm <- 0
    #print( vect_deg[pool] )
    for(i in 1:length(pool))
    {

      somm <- somm + vect_deg[pool[i]]
    }
    if(100 > somm)
    {
      cat("Number of rank down to respect condition=",ref," (SUM of degree=",somm,")\n")
      condition_is_ok <- TRUE
    }
    ref <- ref + 1
  }
  pool
}
```

Then to run the simulation do the following call in the next order:

```r
n <- 501
k <- 3
q <- 2

igraph <- scale_free(n,k,q)
zelot_vector <- zelot(igraph)
igraph <- initZelot(igraph,zelot_vector)
sway_vector <- selection(igraph)


noise <- 0.001
time <- 3000

out <-  0
nb_simu <- 3
winner <- 0
score <- 0
for(i in 1: nb_simu )
{
  out <- simulation(igraph,noise,time,sway_vector,zelot_vector)
  winner <- winner + out[1]
  score <- score + out[2]
  cat("Progress:",i*100/nb_simu,"%\n")
}
cat("TOM win in ",winner/nb_simu,"%\n")
cat("Avreage score is",score/nb_simu,"%\n")
```

With the zealot the effect of the 10 influenced node seem increassed.

c) Effect of remouving node.

As for the nodes, we can influence the network by changing its topology. However, finding the edges to remouve might be more complicated that for nodes in the voter model. Because my proposition of node influencing did not work well, and limitation in time to execute this function: I will not implement it.

TP4

The fourth practical is about epidemic models. We will implement a SIR (Susceptible, Infectious, Recovered) model.

1 R implementation

Below you will find a function that simulate the spread of the epidemic in a SIR meodel. The evolution of the number number of Suceptible, Infected and Recovered can be representated as a function of the time:

```
library(igraph)

plot_epidemic_curves <- function(mat_res,time)
{
  print(mat_res)
  plot( x = c(0:time), y = mat_res[2,],xlab="days",ylab="Percentage of people",type="b",col="green")
  lines(x= c(0:time), y = mat_res[3,],type="b",col="blue")
  lines(x= c(0:time), y = mat_res[1,],type="b",col="red")
  legend( ( time-(time*0.15) ), 90, legend=c("Suceptible", "Infected","Recovered"),
          col=c("green","blue","red"), lty=1:3, cex=0.8)
}

epidemic_plot <- function(igraph,mainstr)
{
  nb_node <- vcount(igraph)
  for(i in 1: nb_node)
  {
    node_status <- get.vertex.attribute(igraph,"epidemic", i)
    if( node_status == "i" )
    {
      igraph <- set.vertex.attribute(igraph, 'color', i, rgb(1,0,0) )
    }
    else if (node_status == "s" )
    {
      igraph <- set.vertex.attribute(igraph, 'color', i, rgb(0,1,0) )
    }
    else if(node_status == "r" )
    {
      igraph <- set.vertex.attribute(igraph, 'color', i, rgb(0,0,1) )
    }
```

```r
    else{
      print("BUG PLOT attribute")
    }
  }
  for(i in 1: nb_node)
  {
    node_status <- get.vertex.attribute(igraph,"i_time", i)

    if( node_status > 0 )
    {
      igraph <- set.vertex.attribute(igraph, "label", i, node_status  )
    }else if( node_status == 0 )
    {
      igraph <- set.vertex.attribute(igraph, "label", i, "R"  )
    }else
    {
      igraph <- set.vertex.attribute(igraph, "label", i, "S"  )
    }


  }
  plot(igraph, main = mainstr )
}

init_epidemic <- function(igraph,n_0,n_d)
{
  nb_node <- vcount(igraph)
  Infected_vect <- floor(runif(n_0,min=1,max=nb_node))

  igraph <- set.vertex.attribute(igraph,"epidemic", value="s")
  igraph <- set.vertex.attribute(igraph,"i_time", value=-1 )
  for( i in 1 : n_0 )
  {
    igraph <- set.vertex.attribute(igraph,"epidemic", Infected_vect[i] ,value="i")
    igraph <- set.vertex.attribute(igraph,"i_time", Infected_vect[i] ,value= n_d )
  }
  epidemic_plot(igraph,"init")
  igraph
}

infected_heal <- function(igraph)
{
  nb_node <- vcount(igraph)
  for( i in 1: nb_node)
  {
    node_time_status <- get.vertex.attribute(igraph,"i_time", i)
    if( node_time_status > 0 )
    {
      #cat("Infected N°",i,"\tremaning time to heal",node_time_status,"\n")
      igraph <- set.vertex.attribute(igraph,"i_time", i ,value= (node_time_status - 1) )
      # node_time_status <- get.vertex.attribute(igraph,"i_time", i)
      # cat("Infected N°",i,"\ttime",node_time_status,"\n")
    }
    if( node_time_status == 0 )
    {
      igraph <- set.vertex.attribute(igraph,"epidemic", i ,value="r")
    }
  }
  igraph
}

transmission <- function(igraph,p_epidemic,n_d)
{
  igraph <- infected_heal(igraph)
  nb_node <- vcount(igraph)

  adj_mat <- as_adjacency_matrix(igraph, type = c("both"))

  vect_i <- vector(mode = "numeric",nb_node)
  vect_i_opo <- vector(mode = "numeric",nb_node)
  for(i in 1:nb_node)
  {
    node_status <- get.vertex.attribute(igraph,"epidemic", i)
    if( node_status == "i" )
    {
      vect_i[i] <- 1
      vect_i_opo[i] <- 0
    }
    else
    {
      vect_i[i] <- 0
      vect_i_opo[i] <- 1
    }
  }

  vect_r_op <- vector(mode = "numeric",nb_node)
  for(i in 1:nb_node)
  {
    node_status <- get.vertex.attribute(igraph,"epidemic", i)
    if( node_status == "r" )
    {
      vect_r_op[i] <- 0
    }
    else
    {
      vect_r_op[i] <- 1
    }
  }


  neigbour_infected_vect <- adj_mat%*%vect_i
  try_infect <- as.vector(neigbour_infected_vect)*vect_r_op*vect_i_opo
  for(i in 1:nb_node)
  {
    if(try_infect[i] > 0 )
    {
      for(j in 1:try_infect[i] )
      {
        if(runif(1) < p_epidemic)
        {
          igraph <- set.vertex.attribute(igraph,"epidemic", i ,value="i")
          igraph <- set.vertex.attribute(igraph,"i_time", i ,value= n_d )
          #cat("We inflect node N°",i,"\n")
```

```r
        }
      }
    }
  }
  igraph
}

stat_days <- function(igraph)
{
  nb_node <- vcount(igraph)
  tot_i <- 0
  tot_s <- 0
  tot_r <- 0
  res <- vector(mode = "numeric")
  for(i in 1:nb_node)
  {
    epi <- get.vertex.attribute(igraph,"epidemic", i )
    if(epi == "i" )
    {
      tot_i <- tot_i + 1
    }
    if(epi == "s" )
    {
      tot_s <- tot_s + 1
    }
    if(epi == "r" )
    {
      tot_r <- tot_r + 1
    }
  }
  res[1] <- tot_i
  res[2] <- tot_s
  res[3] <- tot_r
  res
}

simulation <- function(igraph,p_epidemic,n_d,time)
{
  cat("Begin simulation \n")
  mat_res <- matrix(ncol = time+1,nrow = 3 )
  mat_res[,1 ] <- stat_days(igraph)
  for(i in 1:time )
  {
    cat("Day n°",i,"\n")
    igraph <- transmission(igraph,p_epidemic,n_d)
    restrans <- stat_days(igraph)
    mat_res[, (i+1) ] <- restrans
  }
  mat_res
}


R0_calc <- function(result_mat,nb_node, n_d)
{
  nb_day <- ncol(result_mat)
  gam <- 1/n_d
  outmat <- matrix(nrow = 6, ncol = nb_day)

  x <- result_mat[1,]/nb_node
  s <- result_mat[2,]/nb_node
  r <- result_mat[3,]/nb_node

  d_x <- vector(mode = "numeric",nb_day)
  d_s <- vector(mode = "numeric",nb_day)
  d_r <- vector(mode = "numeric",nb_day)
  beta <- vector(mode = "numeric",nb_day)
  gamma <- vector(mode = "numeric",nb_day)
  for(i in 2:nb_day )
  {
    d_x[i] <- x[i] - x[i-1]
    d_s[i] <- s[i] - s[i-1]
    d_r[i] <- r[i] - r[i-1]
  }
  for(i in 2:nb_day )
  {
    beta[i] <- d_s[i]/(s[i]*x[i])
  }
  print(beta/0.1)
}

n <- 200
p <- 0.01

p_epidemic <- 0.01
time <- 200

n_0 <- 10
n_d <- 10

ig <- Erdos_Renyi_optimized(n,p)
cat("Erdos Renyi created\n")
ig <- init_epidemic(ig,n_0,n_d)
cat("Infection initialized \n")
res <- simulation(ig,p_epidemic,n_d,time)
#R0_calc(res,n,n_d)
res <- res*100/n
plot_epidemic_curves(res,time)
```

$$R_0 = \frac{\beta}{\gamma}$$