Analyse de réseau avec igraph

Hugues Pecout, Laurent Beauguitte, Fernandez Mégane

Analyse de réseau avec `igraph`

Hugues Pecout Laurent Beauguitte Mégane Fernandez

05 avril 2023

Table des matières

Le package `igraph`

Les auteurs et l'autrice de ce tutoriel ont produit ce pdf afin d'assurer un archivage pérenne sur HAL. Si vous souhaitez utiliser ce tutoriel, nous vous recommandons de vous rendre sur la page dédiée du groupe ElémentR.

`igraph` est un logiciel open source principalement écrit en langage C qui fournit des outils pour créer, manipuler, analyser et visualiser des graphes et des réseaux.

La première version a été publiée en 2002 par **Gábor Csárdi** et **Tamás Nepusz**, deux chercheurs en informatique hongrois. Depuis, `igraph` est devenu l'un des packages les plus populaires pour l'analyse de réseau, notamment car cette librairie peut être utilisée dans plusieurs langages de programmation comme **Python**, **Julia** et **C++**.

Gábor Csárdi a officiellement publié l'**interface R pour `igraph` en 2006**. Depuis son adaptation à R, `igraph` a régulièrement été mis à jour pour fournir de nouvelles fonctionnalités et améliorations de performances.

Dans le langage de programmation R, l'interface `igraph` est écrite en R, mais elle utilise des liens (*bindings*) vers les fonctions C fournies par le package `igraph`. Les liens permettent à l'utilisateur d'appeler des fonctions C depuis R, ce qui permet de bénéficier de la rapidité et de l'efficacité de l'implémentation en C d'`igraph`.

`igraph` permet l'**analyse** des **réseaux unimodaux orientés**, **non orientés** et/ou **valués**. Il permet la **transformation** de **réseaux bimodaux** ou **multiplexes** mais ne propose pas ou peu de méthodes pour les analyser. Composée de 809 fonctions, cette librairie offre une grande variété de mesures de réseau, telles que la centralité, la proximité, l'intermédiarité des sommets et des liens, la modularité, la densité, la distance, etc.

Le nombre très élevé de fonctions ne doit pas impressionner : il existe souvent plusieurs fonctions permettant de réaliser la même opération. Ceci alourdit la documentation mais assure une bonne rétro-compatibilité des scripts qui utilisent `igraph`. Un script écrit il y a 5 ou 10 ans fonctionne encore aujourd'hui, ce qui est rare dans R.

`igraph` est également très flexible et peut être utilisé avec d'autres packages R, tels que `ggplot2`, `shiny` ou encore `dplyr` (`tidyverse`).

Installation

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

Documentation

Pour accéder à la documentation interne, utilisez les fonctions suivantes :

```
# Chargement de la librairie
library(igraph)

# Ouverture de l'onglet d'aide
?igraph

# Métadonnées & liste des fonctions
library(help="igraph")
```

Vous pouvez également consulter la documentation officielle du *package* en ligne :

- Site du package igraph (R)
- R igraph manual pages

Citation

Source: inst/CITATION

Csardi G, Nepusz T (2006). "The igraph software package for complex network research." InterJournal, *Complex Systems*, 1695. https://igraph.org.

```
@Article{,
  title = {The igraph software package for complex network research},
  author = {Gabor Csardi and Tamas Nepusz},
  journal = {InterJournal},
  volume = {Complex Systems},
  pages = {1695},
  year = {2006},
  url = {https://igraph.org},
}
```

1 L'objet `igraph`

Le *package* est structuré autour de l'objet `igraph` qui permet de représenter un graphe composé de sommets (nœuds) reliés par des arêtes (liens).

L'objet `igraph` est de type `list`. Tous les éléments qui décrivent un graphe sont ainsi stockés dans un seul et unique objet que l'on va pouvoir manipuler, interroger, analyser et visualiser.

Dans cet objet :

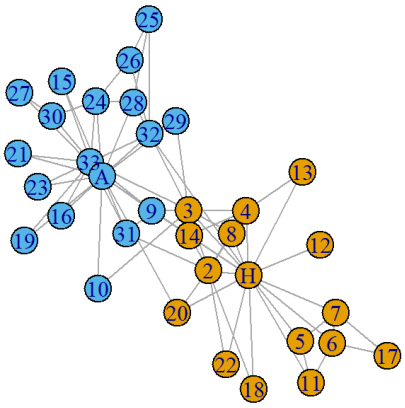
- **les sommets** sont stockés dans l'élément ***vertices*** qui contient leurs noms ;
- **les arêtes** sont stockées dans l'élément ***edges*** qui contient le sommet de départ et d'arrivée de chaque arête ;
- les **attributs** des sommets et des arêtes sont contenus dans les ***vertex.attributes*** et ***edge.attributes***.

Le réseau peut être orienté ou non orienté (*un-directed*), booléen ou valué (*binary* ou *weighted*), unimodal ou biparti (*bipartite*), simple ou avec boucles et/ou liens multiples.

Les différentes fonctions du *package* permettent de manipuler l'objet `igraph` à sa guise. Il s'agit d'une librairie relativement complète que ce support ne fait que survoler.

1.1 Le jeu de données karate

Les données utilisées sont fournies par le package `igraphdata`. Il s'agit du jeu de données ***karate*** qui contient le réseau social entre les membres d'un club de karaté universitaire dirigé par le président John A. et l'instructeur de karaté M. Hi (pseudonymes).



Ces données ont été collectées par l'anthropologue américain Wayne Zachary, d'où le nom de *Zachary karate club* souvent utilisé pour nommer ce jeu de données. Il a observé les relations entre praticiens d'un club de karaté pendant plus de deux ans dans huit contextes sociaux différents. Les matrices de relations ont ensuite été agrégées, ce qui donne un réseau non orienté et valué ; l'intensité des liens correspond au nombre de contextes différents dans lesquels deux praticiens sont en relation. Ce jeu est souvent utilisé pour tester des méthodes de détection de communautés pour une raison simple : lors de son observation, une scission s'est produite et un club dissident a été créé, drainant une partie des élèves.

```
# Chargement de la librairie igraphdata
library(igraphdata)

# Chargement du jeu de données exemple
data(karate)

# Description du jeu de données
?karate
```

1.1.1 Propriétés des arêtes (*edges*)

1.1.2 Propriétés des sommets (*vertices*)

1.2 Construction d'un graphe

Il est possible de construire un objet `igraph` à partir de différents types de données :

- une liste de liens (arêtes) : `graph_from_edgelist()` ;
- une matrice d'adjacence (arêtes) : `graph_from_adjacency_matrix()`;
- un ou deux tableaux (arêtes, arêtes et sommets) : **`graph_from_data_frame()`**.

> **Note**
>
> Le package `igraph` permet également d'importer (et d'exporter) des graphes de différents formats : Pajek, GraphViz, LEDA, GML, GraphML, DIMACS, LGL...

Construire un graphe à partir d'un `data.frame` de liens (arêtes) et d'un `data.frame` contenant les sommets.

```
# Création de l'objet `igraph`
karate <- graph_from_data_frame(d = tableau_aretes,
                                vertices = tableau_sommets,
                                directed = FALSE)
```

Ajouter des attributs à l'ensemble du graphe.

```
# Un nom
graph_attr(karate, "name") <- "Zachary's karate club network"

# Des auteurs
graph_attr(karate, "Author") <- "Wayne W. Zachary"

# Une citation
graph_attr(karate, "Citation") <- "Wayne W. Zachary. An Information Flow Model for Conflic
```

1.3 Interrogation d'un graphe

```
class(karate)
```

```
[1] "igraph"
```

```
summary(karate)
```

```
IGRAPH 4bf6d5a UNW- 34 78 -- Zachary's karate club network
+ attr: name (g/c), Author (g/c), Citation (g/c), name (v/c), label
| (v/c), Faction (v/n), color (v/n), weight (e/n)
```

Description de l'objet `igraph` en trois ou quatre lettres :

D ou U : **graphe orienté**, *Directed* ou *Undirected*
N : ***Named graph***, les sommets possèdent un attribut *name*
W : **pondéré**, les liens possèdent un attribut *weight*.

Lorsqu'un **B** est présent, cela signale un graphe considéré comme bimodal (**bipartite**): les sommets possèdent un attribut *type* (booléen) qui différencie les deux ensembles de sommets.

Les chiffres qui suivent (34 et 78) indiquent le nombre de sommets (ordre) et le nombre d'arêtes (taille).

Chaque attribut est également listé. La première lettre indique à quel élément se rattache l'attribut et la seconde indique le type de variable. Exemple :

(g/c) = Graph/Character
(v/c) = Vertex / Character
(e/n) = Edge / Numeric

La fonction `str()` permet d'avoir une vision plus détaillée du graphe.

```
str(karate)
```

```
Class 'igraph'  hidden list of 10
 $ : num 34
 $ : logi FALSE
 $ : num [1:78] 1 2 3 4 5 6 7 8 10 11 ...
 $ : num [1:78] 0 0 0 0 0 0 0 0 0 0 ...
 $ : num [1:78] 0 1 16 2 17 24 3 4 5 35 ...
```

```
 $ : num [1:78] 0 1 2 3 4 5 6 7 8 9 ...
 $ : num [1:35] 0 0 1 3 6 7 8 11 15 17 ...
 $ : num [1:35] 0 16 24 32 35 37 40 41 41 44 ...
 $ :List of 4
  ..$ : num [1:3] 1 0 1
  ..$ :List of 3
  .. ..$ name    : chr "Zachary's karate club network"
  .. ..$ Author  : chr "Wayne W. Zachary"
  .. ..$ Citation: chr "Wayne W. Zachary. An Information
  Flow Model for Conflict and Fission in Small Groups.
  Journal of Anthropologica"| __truncated__
  ..$ :List of 4
  .. ..$ name   : chr [1:34] "Mr Hi" "Actor 2" "Actor 3" "Actor 4" ...
  .. ..$ label  : chr [1:34] "H" "2" "3" "4" ...
  .. ..$ Faction: num [1:34] 1 1 1 1 1 1 1 2 2 ...
  .. ..$ color  : num [1:34] 1 1 1 1 1 1 1 2 2 ...
  ..$ :List of 1
  .. ..$ weight: num [1:78] 4 5 3 3 3 3 2 2 2 3 ...
 $ :<environment: 0x00000246e554b1e0>
```

Les fonctions `V()` et `E()` permettent de lister les sommets et les arêtes du graphe.

```
# Vertices - Sommets
V(karate)
```

```
+ 34/34 vertices, named, from 4bf6d5a:
[1] Mr Hi  Actor 2  Actor 3  Actor 4  Actor 5  Actor 6  Actor 7  Actor 8
[9] Actor 9  Actor 10 Actor 11 Actor 12 Actor 13 Actor 14 Actor 15 Actor 16
[17] Actor 17 Actor 18 Actor 19 Actor 20 Actor 21 Actor 22 Actor 23 Actor 24
[25] Actor 25 Actor 26 Actor 27 Actor 28 Actor 29 Actor 30 Actor 31 Actor 32
[33] Actor 33 John A
```

```
# Edge - Arêtes
E(karate)
```

```
+ 78/78 edges from 4bf6d5a (vertex names):
[1] Mr Hi--Actor 2  Mr Hi--Actor 3  Mr Hi--Actor 4  Mr Hi--Actor 5
 [5] Mr Hi--Actor 6  Mr Hi--Actor 7  Mr Hi--Actor 8  Mr Hi--Actor 9
 [9] Mr Hi--Actor 11 Mr Hi--Actor 12 Mr Hi--Actor 13 Mr Hi--Actor 14
[13] Mr Hi--Actor 18 Mr Hi--Actor 20 Mr Hi--Actor 22 Mr Hi--Actor 32
[17] Actor 2--Actor 3 Actor 2--Actor 4 Actor 2--Actor 8 Actor 2--Actor 14
```

```
[21] Actor 2--Actor 18 Actor 2--Actor 20 Actor 2--Actor 22 Actor 2--Actor 31
+ ... omitted several edges
```

Comme avec un `data.frame`, on utilise un `$` pour sélectionner un attribut.

```r
# Valeurs de l'attribut "Faction" des sommets
V(karate)$Faction
```

```
 [1] 1 1 1 1 1 1 1 1 2 2 1 1 1 1 2 2 1 1 2 1 2 1 2 2 2 2 2 2 2 2 2 2 2 2
```

```r
# Valeurs de l'attribut "weight" des arêtes
E(karate)$weight
```

```
 [1] 4 5 3 3 3 3 2 2 2 3 1 3 2 2 2 2 6 3 4 5 1 2 2 2 3 4 5 1 3 2 2 2 3 3 3 2 3 5
[39] 3 3 3 3 3 4 2 3 3 2 3 4 1 2 1 3 1 2 3 5 4 3 5 4 2 3 2 7 4 2 4 2 2 4 2 3 3 4
[77] 4 5
```

Lister l'ensemble des attributs des sommets et des arêtes.

```r
# Attributs des sommets
vertex_attr(karate)
```

```
$name
 [1] "Mr Hi" "Actor 2" "Actor 3"  "Actor 4" "Actor 5" "Actor 6"
 [7] "Actor 7" "Actor 8" "Actor 9"  "Actor 10" "Actor 11" "Actor 12"
[13] "Actor 13" "Actor 14" "Actor 15" "Actor 16" "Actor 17" "Actor 18"
[19] "Actor 19" "Actor 20" "Actor 21" "Actor 22" "Actor 23" "Actor 24"
[25] "Actor 25" "Actor 26" "Actor 27" "Actor 28" "Actor 29" "Actor 30"
[31] "Actor 31" "Actor 32" "Actor 33" "John A"

$label
 [1] "H" "2" "3" "4" "5" "6" "7" "8" "9" "10" "11" "12" "13" "14" "15"
[16] "16" "17" "18" "19" "20" "21" "22" "23" "24" "25" "26" "27" "28" "29" "30"
[31] "31" "32" "33" "A"

$Faction
 [1] 1 1 1 1 1 1 1 1 2 2 1 1 1 1 2 2 1 1 2 1 2 1 2 2 2 2 2 2 2 2 2 2 2 2

$color
 [1] 1 1 1 1 1 1 1 1 2 2 1 1 1 1 2 2 1 1 2 1 2 1 2 2 2 2 2 2 2 2 2 2 2 2
```

```
# Attributs des arêtes
edge_attr(karate)
```

```
$weight
 [1] 4 5 3 3 3 3 2 2 2 3 1 3 2 2 2 2 6 3 4 5 1 2 2 2 3 4 5 1 3 2 2 2 3 3 3 2 3 5
[39] 3 3 3 3 3 4 2 3 3 2 3 4 1 2 1 3 1 2 3 5 4 3 5 4 2 3 2 7 4 2 4 2 2 4 2 3 3 4
[77] 4 5
```

1.4 Réseau valué

Pour que le graphe soit considéré comme valué, la table de liens **doit** contenir un attribut (*numeric*) nommé `weight`. C'est le cas dans le jeu de données *karate*.

```
E(karate)$weight
```

```
[1] 4 5 3 3 3 3 2 2 2 3 1 3 2 2 2 2 6 3 4 5 1 2 2 2 3 4 5 1 3 2 2 2 3 3 3 2 3 5
[39] 3 3 3 3 3 4 2 3 3 2 3 4 1 2 1 3 1 2 3 5 4 3 5 4 2 3 2 7 4 2 4 2 2 4 2 3 3 4
[77] 4 5
```

```
# Fonction pour savoir si le graphe est valué
is_weighted(karate)
```

```
[1] TRUE
```

1.5 Modification d'un graphe

Ajouter des sommets.

```
karate <- add_vertices(graph = karate,
                       nv = 1,
                       name = "Toto",
                       label = "35",
                       Faction = 1,
                       color = 1)
```

Ajouter des arêtes.

```r
karate <- add_edges(graph = karate,
                    edges = c("Mr Hi", "Toto", "Toto", "Actor 2"),
                    attr = list(weight = 5))

summary(karate)
```

```
IGRAPH 4c0af0a UNW- 35 80 -- Zachary's karate club network
+ attr: name (g/c), Author (g/c), Citation (g/c), name (v/c), label
| (v/c), Faction (v/n), color (v/n), weight (e/n)
```

Modifier ou supprimer les attributs des sommets ou des arêtes.

Les arguments à renseigner pour les modifications sont `graph` (nom du graphe), `name` (nom de l'attribut à modifier), `index` (sous-ensemble d'arêtes que l'on veut modifier, optionnel), `value` (nouvelle valeur de l'attribut pour toutes les arêtes ou pour les arêtes du sous-ensemble défini dans `index`).

```r
# Suppression
delete_edge_attr()
delete_vertex_attr()

# Modification
set_edge_attr()
set_vertex_attr()
```

Créer un attribut.

Les valeurs doivent respecter l'ordre des sommets de l'objet `igraph`.

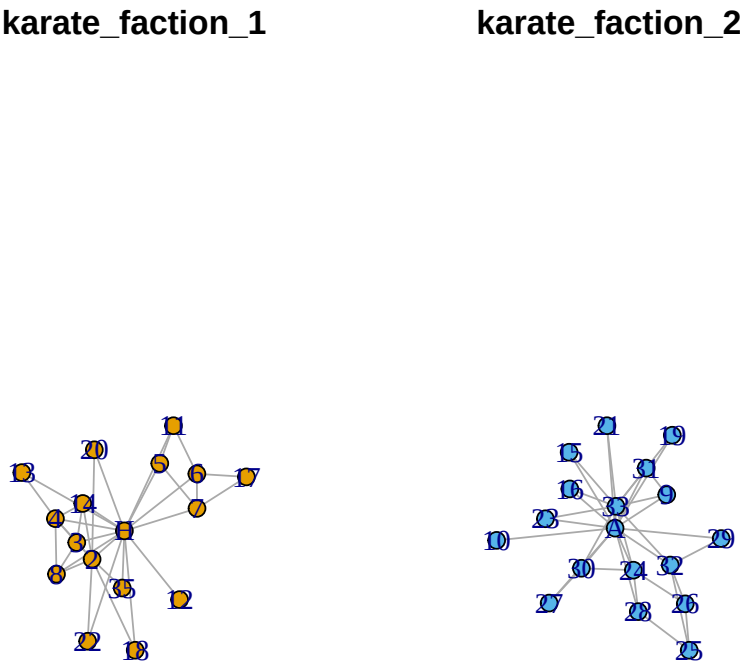
```r
V(karate)$position <-  c("Maître", "Elève", "Elève", "Elève", "Elève", "Elève", "Elève", "
                         "Elève", "Elève", "Elève",  "Elève", "Elève", "Elève", "Elève", "
                         "Elève", "Elève", "Elève", "Elève", "Elève", "Elève", "Elève", "E
                         "Elève", "Elève", "Elève", "Elève", "Elève", "Elève", "Maître", "
```

1.6 Extraction d'un sous-graphe

Extraire des sous-graphes à partir d'un attribut des sommets.

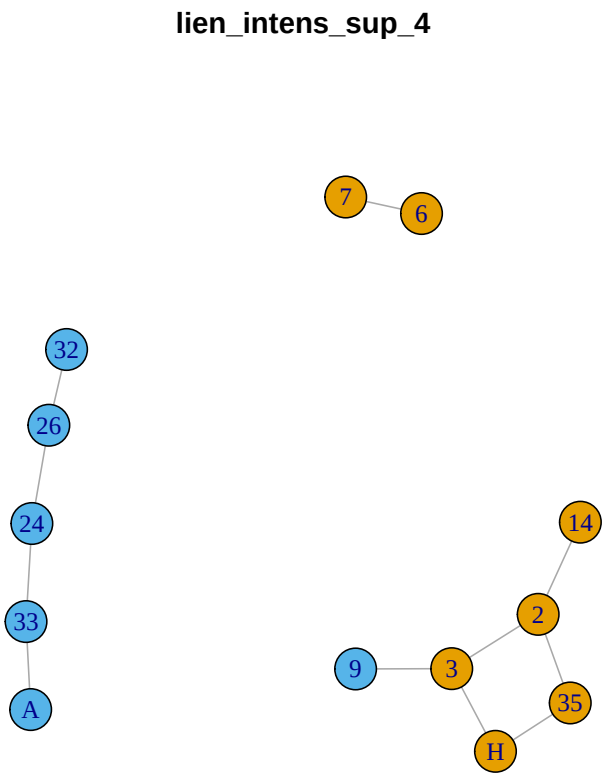
```r
# Sélection des membres de la faction 1
karate_faction_1 <- induced_subgraph(karate, v = which(V(karate)$Faction == 1))
```

```r
# Sélection des membres de la faction 2
karate_faction_2 <- induced_subgraph(karate, v = which(V(karate)$Faction == 2))
```



Extraire un sous-graphe en fonction d'un attribut des liens.

```r
# Sélection des liens d'intensité supérieure ou égale à 4
lien_intens_sup_4 <- subgraph.edges(karate, eids = which(E(karate)$weight > 4))
```



1.7 Extraction des ego-networks

La fonction `make_ego_graph` permet d'extraire les ego-networks d'un graphe. L'argument `order` permet de choisir le niveau de voisinage à prendre en compte (1 par défaut). L'argument `mode` permet de prendre en compte tous les liens ("all"), uniquement les liens entrants ("in") ou uniquement les liens sortants ("out") dans le cas d'un réseau orienté.

```r
# Création d'une liste d'objets igraph (tous les ego-networks)
EgoNet_karate <- make_ego_graph(karate)

# Paramétrage de l'ego network
EgoNet_karate <- make_ego_graph(karate,
                                nodes = V(karate),
                                order = 1,
                                mode = c("all"))

# Sélectionner un ego-network dans la liste - V(karate)$name[1]
EgoNet_karate[[1]] # ici le premier
```

```
IGRAPH 4c234d1 UNW- 18 36 -- Zachary's karate club network
+ attr: name (g/c), Author (g/c), Citation (g/c), name (v/c), label
| (v/c), Faction (v/n), color (v/n), position (v/c), weight (e/n)
+ edges from 4c234d1 (vertex names):
 [1] Mr Hi  --Actor 2  Mr Hi  --Actor 3  Mr Hi  --Actor 4  Mr Hi  --Actor 5
 [5] Mr Hi  --Actor 6  Mr Hi  --Actor 7  Mr Hi  --Actor 8  Mr Hi  --Actor 9
 [9] Mr Hi  --Actor 11 Mr Hi  --Actor 12 Mr Hi  --Actor 13 Mr Hi  --Actor 14
[13] Mr Hi  --Actor 18 Mr Hi  --Actor 20 Mr Hi  --Actor 22 Mr Hi  --Actor 32
[17] Actor 2--Actor 3  Actor 2--Actor 4  Actor 2--Actor 8  Actor 2--Actor 14
[21] Actor 2--Actor 18 Actor 2--Actor 20 Actor 2--Actor 22 Actor 3--Actor 4
[25] Actor 3--Actor 8  Actor 3--Actor 9  Actor 3--Actor 14 Actor 4--Actor 8
+ ... omitted several edges
```

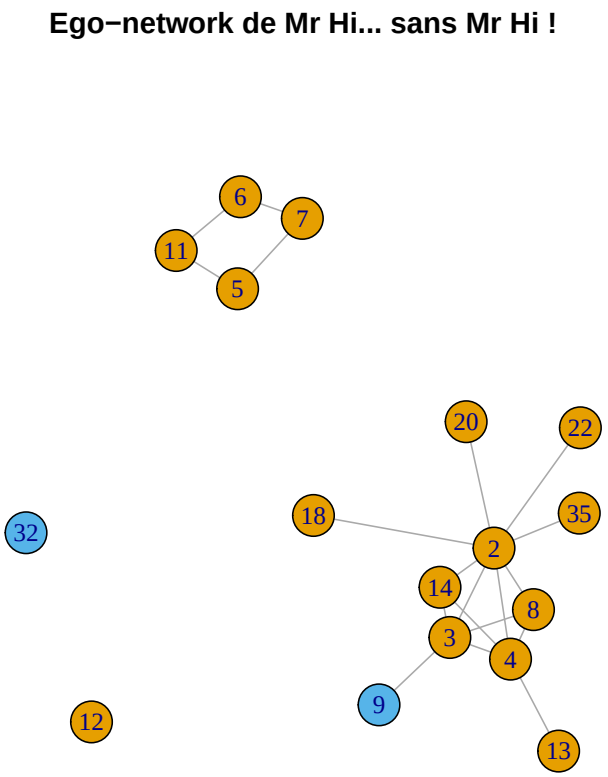
```r
# Création d'un ego-network à partir d'un nom (crée une liste avec un ego-network)
EgoNet_JohnA <- make_ego_graph(karate,
                               nodes = V(karate)[name=='John A'], # on peut aussi faire la
                               order = 1,
                               mode = c("all"))
EgoNet_JohnA[[1]]
```

```
IGRAPH 4c248e1 UNW- 18 32 -- Zachary's karate club network
+ attr: name (g/c), Author (g/c), Citation (g/c), name (v/c), label
| (v/c), Faction (v/n), color (v/n), position (v/c), weight (e/n)
+ edges from 4c248e1 (vertex names):
 [1] Actor 9 --Actor 31 Actor 9 --Actor 33 Actor 9 --John A   Actor 10--John A
 [5] Actor 14--John A   Actor 15--Actor 33 Actor 15--John A   Actor 16--Actor 33
 [9] Actor 16--John A   Actor 19--Actor 33 Actor 19--John A   Actor 20--John A
[13] Actor 21--Actor 33 Actor 21--John A   Actor 23--Actor 33 Actor 23--John A
```

```
[17] Actor 24--Actor 28 Actor 24--Actor 30 Actor 24--Actor 33 Actor 24--John A
[21] Actor 27--Actor 30 Actor 27--John A   Actor 28--John A   Actor 29--Actor 32
[25] Actor 29--John A   Actor 30--Actor 33 Actor 30--John A   Actor 31--Actor 33
+ ... omitted several edges
```

Supprimer ego de son réseau personnel.

```
Karate_sans_Hi <- induced_subgraph(EgoNet_karate[[1]],
                                   v = which(V(EgoNet_karate[[1]])$name != c("Mr Hi")))
```



S'il est facile de créer les réseaux personnels à partir d'un réseau complet, `igraph` ne propose pas les mesures spécifiques développées pour l'analyse des réseaux personnels.

1.8 Boucles et liens multiples

`igraph` peut prendre en charge des réseaux avec boucles (liens d'un sommet vers lui-même) et liens multiples ; certaines mesures fonctionnent sans problème avec ce type de réseau.

La fonction `simplify()` permet de transformer ces réseaux en réseaux simples. La fonction `simplify_and_colorize()` permet de garder en mémoire l'existence de ces boucles et liens multiples.

```
# Manipulation d'un réseau avec boucle et liens multiples
gbm <- d1 <- rbind(c("A","A"),
                   c("A","B"),
                   c("B","C"),
                   c("B","C"),
                   c("B","C"),
                   c("C","D"),
                   c("C","D"))


g1 <- graph.data.frame(d1, directed = FALSE)
g1
```

```
IGRAPH 4c2f2dc UN-- 4 7 --
+ attr: name (v/c)
+ edges from 4c2f2dc (vertex names):
[1] A--A A--B B--C B--C B--C C--D C--D
```

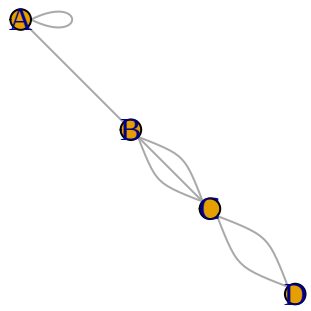
```
is.loop(g1)
```

```
[1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
```

```
is.multiple(g1)
```

```
[1] FALSE FALSE FALSE  TRUE  TRUE FALSE  TRUE
```

```
plot(g1)
```




```
degree(g1)
```

```
A B C D
3 4 5 2
```

La fonction `degree()` donne un résultat satisfaisant. Le sommet A a un degré de 3 : un lien avec B et une boucle. Par convention, une boucle est considérée comme un lien entrant plus un lien sortant et le degré d'un sommet avec boucle est donc toujours supérieur ou égal à 2.

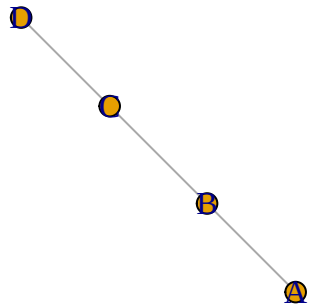
```
graph.density(g1)
```

```
[1] 1.166667
```

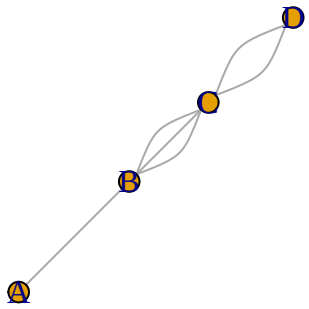
Par contre, la densité donne un résultat logique mais peu satisfaisant. Il considère qu'il y a 7 liens présents et 6 liens possibles (4*3/2) d'où une densité supérieure à 1. Or, le nombre de liens possibles est de 10 (si on considère que des boucles peuvent être partout présentes) voire de 22 (3 liens possibles entre chaque paire de sommets donc 3 fois 6, plus 4 boucles).

L'approche standard en analyse de réseau avec ce type d'objet est de commencer par supprimer boucles et liens multiples.

```r
# Par défaut, simplify supprimer boucles et liens multiples
g1_2 <- simplify(g1)
plot(g1_2)
```




```r
# Si je veux garder les liens multiples
g1_3 <- simplify(g1, remove.multiple = FALSE)
plot(g1_3)
```




```
# Si je veux garder une trace des boucles et des liens multiples
g1_4 <- simplify_and_colorize(g1)
g1_4
```

```
IGRAPH 4c3847a U--- 4 3 --
+ attr: color (v/n), color (e/n)
+ edges from 4c3847a:
[1] 1--2 2--3 3--4
```

```
V(g1_4)$color # il y avait une boucle au premier sommet
```

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

```
E(g1_4)$color # il y avait 1 lien, 3 liens, 2 liens
```

```
[1] 1 3 2
```

1.9 Réseau bimodal (ou biparti)

Pour qu'un réseau soit considéré comme biparti, il faut que la table des sommets contienne un attribut booléen nommé `type` pour différencier les deux catégories de sommets. Cette condition est nécessaire et suffisante.

```
# Fonction pour savoir si le graphe est biparti
is.bipartite(karate)
```

```
[1] FALSE
```

```
# Création de l'attribut "type"
# Valeur booléenne pour dissocier les deux groupes
V(karate)$type <- ifelse(V(karate)$color == 1, TRUE, FALSE)

# La présence de l'attribut "type" rend l'objet igraph biparti
is.bipartite(karate)
```

```
[1] TRUE
```

Comme il y a un attribut `type` booléen sur les sommets, `igraph` considère que le réseau est bimodal alors que la liste des liens montre qu'il ne l'est pas. Il ne sera pas possible, et c'est normal, de transformer ce faux réseau bimodal.

Le script ci-dessous crée un mini réseau bimodal. `igraph` permet peu d'analyse. Si le degré donne un résultat correct, le degré normalisé est faux, tout comme la densité.

```
# Création d'un réseau minimal
som <- c("a1","a2","a3","a4","A1","A2","A3")
somtype <- c(0,1,0,1,1,0,0)
gb <- make_graph(c("a1","A1","a2","A1",
                   "a2","A2","a2","A3",
                   "a3","A2","a4","A3",
                   "a4", "A2"), directed = FALSE)

gb
```

```
IGRAPH 4c3f8ea UN-- 7 7 --
+ attr: name (v/c)
+ edges from 4c3f8ea (vertex names):
[1] a1--A1 A1--a2 a2--A2 a2--A3 A2--a3 A3--a4 A2--a4
```

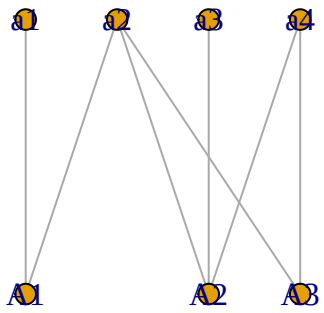
Il n'y a pas de variable attribut booléen 'type', le réseau est donc considéré comme un réseau unimodal.

```
V(gb)$type <- somtype

gb
```

```
IGRAPH 4c3f8ea UN-B 7 7 --
+ attr: name (v/c), type (v/n)
+ edges from 4c3f8ea (vertex names):
[1] a1--A1 A1--a2 a2--A2 a2--A3 A2--a3 A3--a4 A2--a4
```

*UN-**B*** signale que le réseau est dorénavant considéré comme **biparti**.

```
plot(gb, layout = layout_as_bipartite(gb))
```




```
# Degré : résultat correct
degree(gb)
```

```
a1 A1 a2 A2 A3 a3 a4
 1  2  3  3  2  1  2
```

```r
# Degré nortmalisé : résultat faux
degree(gb, normalized = TRUE)
```

```
       a1        A1        a2        A2        A3        a3        a4
0.1666667 0.3333333 0.5000000 0.5000000 0.3333333 0.1666667 0.3333333
```

```r
# Densité : résultat faux
graph.density(gb)
```

```
[1] 0.3333333
```

Dans un réseau bimodal $G = \{V_1, V_2, E\}$, le degré maximum d'un sommet de l'ensemble V_1 est égal au nombre de sommets de l'ensemble V_2. Pour normaliser, on divise par le nombre du sommets de l'autre ensemble. Ici par exemple, *a1* a un degré de 1, le maximum est 3, son degré normalisé est donc de 1/3.

En ce qui concerne la densité, le nombre maximum de liens possibles est égal à $V_1 \times V_2$ soit 12 dans cet exemple. Il y a 7 liens, la densité est donc de 0.58.

L'une des seules options disponibles est la possibilité de le projeter pour créer deux réseaux unimodaux valués.

Si je considère que les lettres majuscules sont des autrices publiant dans des revues (lettres minuscules), un lien entre deux revues (réseau du milieu) indique combien ces revues partagent d'autrices en commun. Un lien entre deux autrices dans le réseau valué de droite indiquent dans combien de revues communes elles ont publié.

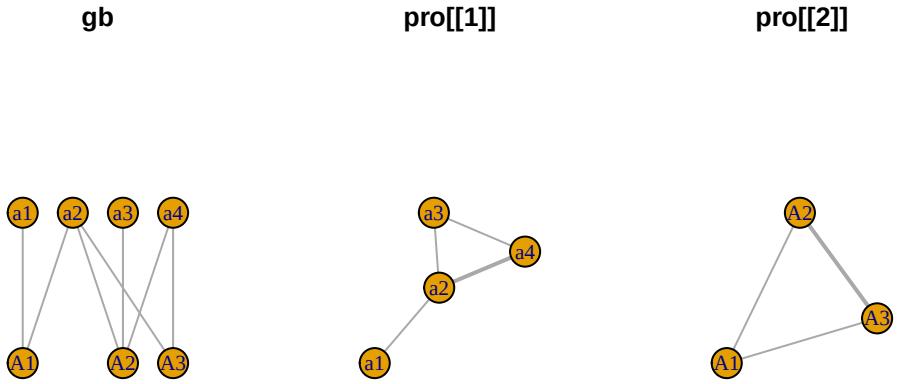
```r
# Projections du réseau bimodal
pro <- bipartite.projection(gb)

# Intensité des liens dans la première projection
E(pro[[1]])$weight
```

```
[1] 1 1 2 1
```

```r
# Intensité des liens dans la deuxième projection
E(pro[[2]])$weight
```

```
[1] 1 1 2
```

Visualisation du réseau biparti et des deux projections qui en découlent :



1.10 Exporter un objet `igraph`

Extraire les tableaux des sommets et des arêtes avec leur attributs.

```
# Extraction des liens et de leurs attributs
tableau_aretes <- as_data_frame(karate, what="edges")

# Extraction des sommets et de leurs attributs
tableau_sommets <- as_data_frame(karate, what="vertices")
```

2 Indicateurs globaux

2.1 Nombre de sommets et d'arêtes

Calculer le nombre de sommets (ordre) et d'arêtes (taille) du graphe.

```
# Nombre de sommets
vcount(karate)
```

```
[1] 34
```

```
# Nombre d'arêtes
ecount(karate)
```

```
[1] 78
```

2.2 Densité

La densité d'un graphe désigne le rapport entre le nombre de liens présents et le nombre de liens possibles. Elle varie entre 0 (graphe vide, aucun lien) et 1 (graphe complet, tous les sommets sont directement reliés entre eux).

```
# Densité d'un graphe
graph.density(karate)
```

```
[1] 0.1390374
```

2.3 Composantes connexes

Un graphe est dit connexe s'il existe au moins un chemin entre toutes les paires de sommets. Dans le cas contraire, le graphe est constitué de différentes composantes connexes.

```r
# Composantes connexes d'un graphe
components(karate)
```

```
$membership
   Mr Hi  Actor 2  Actor 3  Actor 4  Actor 5  Actor 6  Actor 7  Actor 8
       1        1        1        1        1        1        1        1
 Actor 9 Actor 10 Actor 11 Actor 12 Actor 13 Actor 14 Actor 15 Actor 16
       1        1        1        1        1        1        1        1
Actor 17 Actor 18 Actor 19 Actor 20 Actor 21 Actor 22 Actor 23 Actor 24
       1        1        1        1        1        1        1        1
Actor 25 Actor 26 Actor 27 Actor 28 Actor 29 Actor 30 Actor 31 Actor 32
       1        1        1        1        1        1        1        1
Actor 33   John A
       1        1

$csize
[1] 34

$no
[1] 1
```

La fonction crée une liste composée de trois éléments :

- **membership** : identifiant de la composante connexe d'appartenance ;
- **csize** : nombre de sommets des composantes connexes ;
- **no** : nombre de composantes connexes.

Extraire toute les composantes connexes.

```r
Mes_composantes <- decompose(karate)

# Cela construit un objet List d'objets igraph
Mes_composantes
```

```
[[1]]
IGRAPH 4d6a973 UNW- 34 78 -- Zachary's karate club network
+ attr: name (g/c), Citation (g/c), Author (g/c), Faction (v/n), name
| (v/c), label (v/c), color (v/n), weight (e/n)
+ edges from 4d6a973 (vertex names):
 [1] Mr Hi  --Actor 2  Mr Hi  --Actor 3  Mr Hi  --Actor 4  Mr Hi  --Actor 5
 [5] Mr Hi  --Actor 6  Mr Hi  --Actor 7  Mr Hi  --Actor 8  Mr Hi  --Actor 9
```

```
 [9] Mr Hi  --Actor 11 Mr Hi  --Actor 12 Mr Hi  --Actor 13 Mr Hi  --Actor 14
[13] Mr Hi  --Actor 18 Mr Hi  --Actor 20 Mr Hi  --Actor 22 Mr Hi  --Actor 32
[17] Actor 2--Actor 3  Actor 2--Actor 4  Actor 2--Actor 8  Actor 2--Actor 14
[21] Actor 2--Actor 18 Actor 2--Actor 20 Actor 2--Actor 22 Actor 2--Actor 31
[25] Actor 3--Actor 4  Actor 3--Actor 8  Actor 3--Actor 9  Actor 3--Actor 10
+ ... omitted several edges
```

```r
# Accéder à chaque objet igraph créé
composante_1 <- Mes_composantes[[1]]
```

2.4 Diamètre

Le diamètre d'un graphe est le plus long des plus courts chemins du graphe.

```r
# Diamètre d'un graphe (valué)
diameter(karate)
```

```
[1] 13
```

Par défaut, la fonction `diameter()` prend en compte le poids (*weight*) des liens.

```r
# Diamètre d'un graphe (topologique)
diameter(karate, directed = FALSE, weights = NA)
```

```
[1] 5
```

> **Note**
>
> Si le graphe possède plusieurs composantes connexes, la fonction renvoie le plus grand diamètre.

Pour connaître les sommets du diamètre.

```r
get_diameter(karate, weights = NA)
```

```
+ 6/34 vertices, named, from 4b458a1:
[1] Actor 15 Actor 33 Actor 3  Mr Hi    Actor 6  Actor 17
```

2.5 Transitivité globale

La transitivité d'un graphe mesure la proportion de triades fermées (clique de trois sommets). Ce coefficient mesure à quel point le voisinage d'un sommet est connecté.

```
transitivity(karate, type = "global")
```

```
[1] 0.2556818
```

3 Mesures locales

Les mesures disponibles sont nombreuses, nous avons listé ici les plus couramment employées.

3.1 Centralité de degré

Nombre de liens adjacents à un sommet.

```
degree(karate)
```

```
   Mr Hi  Actor 2  Actor 3  Actor 4  Actor 5  Actor 6  Actor 7  Actor 8
      16        9       10        6        3        4        4        4
 Actor 9 Actor 10 Actor 11 Actor 12 Actor 13 Actor 14 Actor 15 Actor 16
       5        2        3        1        2        5        2        2
Actor 17 Actor 18 Actor 19 Actor 20 Actor 21 Actor 22 Actor 23 Actor 24
       2        2        2        3        2        2        2        5
Actor 25 Actor 26 Actor 27 Actor 28 Actor 29 Actor 30 Actor 31 Actor 32
       3        3        2        4        3        4        4        6
Actor 33   John A
      12       17
```

3.1.1 Degré normalisé

L'argument `normalized` permet de calculer la centralité de degré normalisée (*degré / (nombre de sommets-1)*). Quand on étudie un réseau orienté, l'argument `mode` permet de calculer le degré entrant ("in"), sortant ("out") ou les deux ("all").

```
degree(karate, normalized = TRUE, mode = "all")
```

```
     Mr Hi     Actor 2     Actor 3     Actor 4     Actor 5     Actor 6     Actor 7
0.48484848 0.27272727 0.30303030 0.18181818 0.09090909 0.12121212 0.12121212
   Actor 8     Actor 9    Actor 10    Actor 11    Actor 12    Actor 13    Actor 14
0.12121212 0.15151515 0.06060606 0.09090909 0.03030303 0.06060606 0.15151515
```

```
  Actor 15   Actor 16   Actor 17   Actor 18   Actor 19   Actor 20   Actor 21
0.06060606 0.06060606 0.06060606 0.06060606 0.06060606 0.09090909 0.06060606
  Actor 22   Actor 23   Actor 24   Actor 25   Actor 26   Actor 27   Actor 28
0.06060606 0.06060606 0.15151515 0.09090909 0.09090909 0.06060606 0.12121212
  Actor 29   Actor 30   Actor 31   Actor 32   Actor 33     John A
0.09090909 0.12121212 0.12121212 0.18181818 0.36363636 0.51515152
```

3.1.2 Degré pondéré

Degré pondéré par l'intensité des liens (attribut *weight*).

```
strength(karate)
```

```
   Mr Hi  Actor 2  Actor 3  Actor 4  Actor 5  Actor 6  Actor 7  Actor 8
      42       29       33       18        8       14       13       13
 Actor 9 Actor 10 Actor 11 Actor 12 Actor 13 Actor 14 Actor 15 Actor 16
      17        3        8        3        4       17        5        7
Actor 17 Actor 18 Actor 19 Actor 20 Actor 21 Actor 22 Actor 23 Actor 24
       6        3        3        5        4        4        5       21
Actor 25 Actor 26 Actor 27 Actor 28 Actor 29 Actor 30 Actor 31 Actor 32
       7       14        6       13        6       13       11       21
Actor 33   John A
      38       48
```

3.2 Distribution des degrés

La fonction `degree.distribution()` permet de calculer la fréquence relative des sommets par degré.

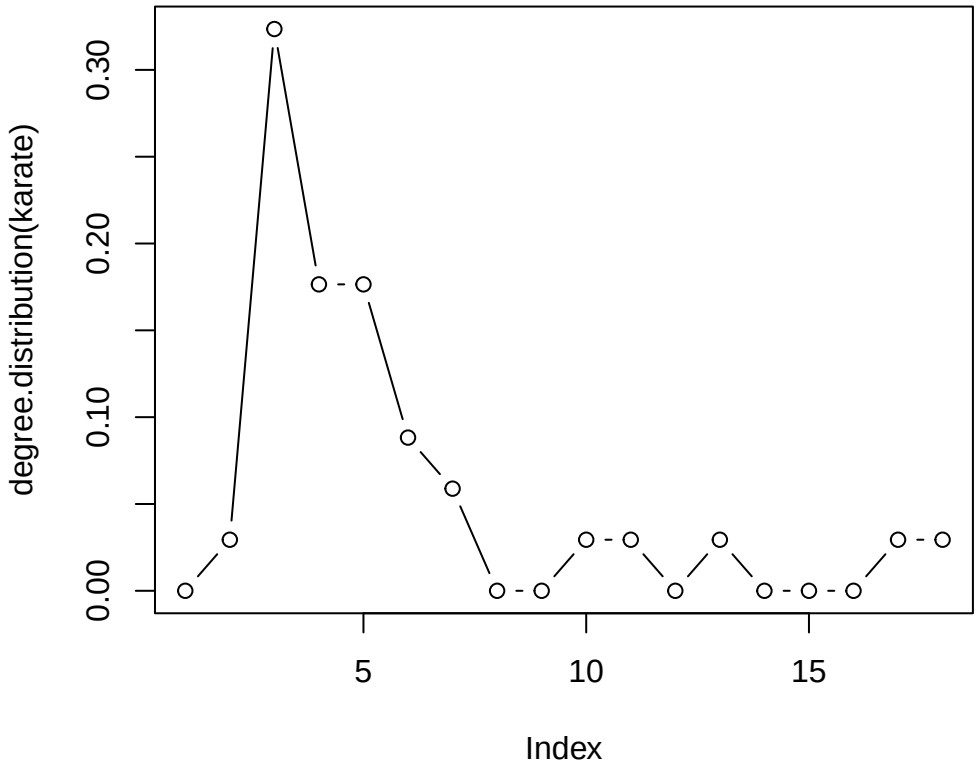
```
# mode = c("all", "out", "in", "total")
# loops = TRUE/FALSE
# cumulative = TRUE/FALSE

degree.distribution(karate)
```

```
 [1] 0.00000000 0.02941176 0.32352941 0.17647059 0.17647059 0.08823529
 [7] 0.05882353 0.00000000 0.00000000 0.02941176 0.02941176 0.00000000
[13] 0.02941176 0.00000000 0.00000000 0.00000000 0.02941176 0.02941176
```

- 1ère valeur = fréquence relative des sommets de degré 0
- 2ème valeur = fréquence relative des sommets de degré 1
- ...

Représenter la distribution des degrés.

```
plot(degree.distribution(karate), type="b")
```



Aucun sommet ne présente un degré égal à 0, il n'y a donc aucun sommet isolé.

3.3 Les sommets voisins

Pour connaître les voisins d'un sommets, utiliser la fonction `neighbors()`.

```
neighbors(karate, v = 1)
```

```
+ 16/34 vertices, named, from 4b458a1:
 [1] Actor 2  Actor 3  Actor 4  Actor 5  Actor 6  Actor 7  Actor 8  Actor 9
 [9] Actor 11 Actor 12 Actor 13 Actor 14 Actor 18 Actor 20 Actor 22 Actor 32
```

Pour récupérer la liste des liens vers les sommets voisins, utiliser la fonction `incident()`.

```
incident(karate, v = 1)
```

```
+ 16/78 edges from 4b458a1 (vertex names):
 [1] Mr Hi--Actor 2  Mr Hi--Actor 3  Mr Hi--Actor 4  Mr Hi--Actor 5
 [5] Mr Hi--Actor 6  Mr Hi--Actor 7  Mr Hi--Actor 8  Mr Hi--Actor 9
 [9] Mr Hi--Actor 11 Mr Hi--Actor 12 Mr Hi--Actor 13 Mr Hi--Actor 14
[13] Mr Hi--Actor 18 Mr Hi--Actor 20 Mr Hi--Actor 22 Mr Hi--Actor 32
```

Pour savoir si deux sommets sont connectés, utiliser la fonction `are.connected()`.

```
are.connected(karate, v1 = 1, v2 = 34)
```

```
[1] FALSE
```

3.4 Centralité d'intermédiarité

Elle calcule la proportion de plus courts chemins passant par un sommet donné.

```
betweenness(karate, weights = NULL, normalized=TRUE, directed = FALSE)
```

```
      Mr Hi      Actor 2      Actor 3      Actor 4      Actor 5      Actor 6
0.4737689394 0.0640151515 0.0694128788 0.0025252525 0.0009469697 0.0293560606
     Actor 7      Actor 8      Actor 9     Actor 10     Actor 11     Actor 12
0.0293560606 0.0000000000 0.0248106061 0.0137941919 0.0009469697 0.0000000000
    Actor 13     Actor 14     Actor 15     Actor 16     Actor 17     Actor 18
0.0000000000 0.0022727273 0.0000000000 0.0000000000 0.0000000000 0.0304924242
    Actor 19     Actor 20     Actor 21     Actor 22     Actor 23     Actor 24
0.0056818182 0.2406565657 0.0000000000 0.0000000000 0.0000000000 0.0018939394
    Actor 25     Actor 26     Actor 27     Actor 28     Actor 29     Actor 30
```

```
0.0640782828 0.0009469697 0.0000000000 0.0123106061 0.0191287879 0.0000000000
    Actor 31     Actor 32     Actor 33       John A
0.0056818182 0.1256313131 0.0722222222 0.3967803030
```

Les arguments `normalized` et `weights` permettent de normaliser ou non la centralité d'intermédiarité et d'utiliser ou non le poids des arêtes.

3.5 Centralité de proximité

Cet indicateur est l'inverse de la somme des distances des plus courts chemins d'un sommet vers tous les autres, il varie entre 0 et 1.

```
closeness(karate, weights = NULL, normalized=TRUE, mode = "all")
```

```
    Mr Hi    Actor 2    Actor 3    Actor 4    Actor 5    Actor 6    Actor 7    Actor 8
0.2538462 0.2000000 0.1964286 0.1764706 0.1527778 0.1520737 0.1534884 0.1823204
  Actor 9  Actor 10  Actor 11  Actor 12  Actor 13  Actor 14  Actor 15  Actor 16
0.1987952 0.1907514 0.1755319 0.1460177 0.2049689 0.1907514 0.1709845 0.1375000
 Actor 17  Actor 18  Actor 19  Actor 20  Actor 21  Actor 22  Actor 23  Actor 24
0.1085526 0.1929825 0.1875000 0.2481203 0.2037037 0.1764706 0.1586538 0.1386555
 Actor 25  Actor 26  Actor 27  Actor 28  Actor 29  Actor 30  Actor 31  Actor 32
0.1578947 0.1235955 0.1692308 0.1563981 0.2024540 0.1746032 0.1736842 0.2088608
 Actor 33     John A
0.2000000 0.2519084
```

> **Note**
>
> **A.** Calculer les centralités sur chaque composante connexe.
> **Si votre réseau n'est pas connexe, il est nécessaire d'utiliser la fonction `decompose` puis de mesurer les centralités d'intermédiarité et de proximité dans chaque composante**. Sinon, le résultat fourni par `igraph` est aberrant ; attention, il n'y a ni warning ni message d'erreur...
>
> ```
> betweenness(Mes_composantes[[1]], weights = NULL, normalized=TRUE, directed = FALSE)
>
> closeness(Mes_composantes[[1]], weights = NULL, normalized=TRUE, mode = "all")
> ```
>
> **B.** Mesurer la centralité d'un ou de plusieurs sommets en particulier.

```r
# Par l'index
degree(karate)[1]
```

```
Mr Hi
   16
```

```r
# Par le nom du sommet
betweenness(karate, weights=NULL, normalized=TRUE)["Mr Hi"]
```

```
    Mr Hi
0.4737689
```

```r
# Sélection multiple
closeness(karate, weights=NULL, normalized=TRUE)[c(1,34)]
```

```
    Mr Hi    John A
0.2538462 0.2519084
```

3.6 Transitivité locale

Mesurer la `transitivity()` locale (par sommet) avec l'argument `type`.

```r
transitivity(karate, type = c("local"))
```

```
 [1] 0.1500000 0.3333333 0.2444444 0.6666667 0.6666667 0.5000000 0.5000000
 [8] 1.0000000 0.5000000 0.0000000 0.6666667       NaN 1.0000000 0.6000000
[15] 1.0000000 1.0000000 1.0000000 1.0000000 1.0000000 0.3333333 1.0000000
[22] 1.0000000 1.0000000 0.4000000 0.3333333 0.3333333 1.0000000 0.1666667
[29] 0.3333333 0.6666667 0.5000000 0.2000000 0.1969697 0.1102941
```

> **Note**
>
> Il n'est pas possible de calculer la transitivité pour les sommets de degré inférieur à deux : par définition, ils ne sont membres d'aucune triade. La fonction renvoie alors la valeur '**NaN**' (**N**ot **a** **N**umber).

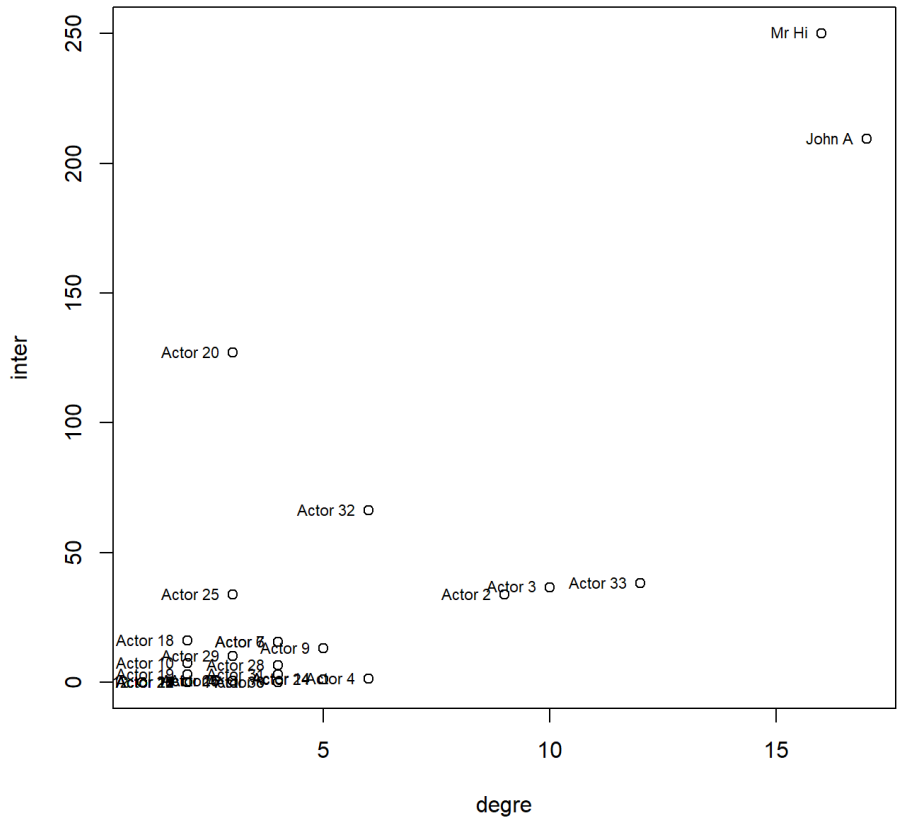
3.7 Relations entre indicateurs

La fonction `plot()` permet de visualiser la relation entre deux indicateurs de centralité. Si le graphique suggère une relation linéaire, on peut utiliser la fonction `cor.test` (corrélation).

```
degre <- degree(karate)
inter <- betweenness(karate)
```

Représentation graphique.

```
# Représentation graphique
plot(x = degre, y = inter)
text(degre, inter, labels = V(karate)$name,cex = 0.7, pos = 2)
```



Mesure de la corrélation avec `cor.test()`.

```
# Méthode Pearson par défaut
cor.test(degre, inter)
```

```
    Pearson's product-moment correlation

data:  degre and inter
t = 7.2451, df = 32, p-value = 3.128e-08
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
 0.6135820 0.8893245
sample estimates:
      cor
```

```
0.7882018
```

t : valeur de la statistique
df : degrés de liberté
p-value : p-value
cor : coefficient de corrélation (de Pearson dans ce cas).

3.8 Les mesures comme attributs

Il est possible de transformer les différentes mesures en attributs des sommets et/ou des liens.

```r
# Centralité de degré
V(karate)$degre <- degree(karate)

# Centralité d'intermediarité
V(karate)$intermed <- betweenness(karate, weights=NULL, normalized=TRUE)

# Centralité de proximité
V(karate)$proximit <- closeness(karate, weights=NULL, normalized=TRUE)

# Transitivité locale
V(karate)$transit_loc <- transitivity(karate, type = c("local"))
```

Les sommets ont de nouveaux attributs.

4 Cliques et communautés

4.1 Détection de cliques

Une clique est un sous-graphe maximal complet, c'est-à-dire le plus grand nombre possible de sommets entre lesquels tous les liens possibles sont présents. La fonction `clique` du *package* `igraph` ne prend pas en compte la notion de sous-graphe maximal et fournit par défaut toutes les "cliques" d'ordre 2 (i.e. tous les liens), d'ordre 3 (triades fermées), d'ordre 4, etc. etc. Il est donc recommandé d'utiliser l'argument optionnel `min` pour indiquer le nombre minimum de sommets devant constituer une clique ; cet argument est fonction de la taille et de la densité du réseau étudié. Par ailleurs, que votre réseau soit ou non orienté, la fonction ne prend pas en compte l'orientation des liens.

Si l'on cherche des motifs précis (par exemple des sous-graphes maximaux complets d'ordre n), on peut fixer le paramètre `max` en lui donnant comme argument la même valeur que pour `min`. Les lignes ci-dessous permettent de récupérer toutes les triades fermées.

```
# Lister toutes les cliques de 3 sommets
Cliques_3 <- cliques(karate, min = 3, max = 3)

# Affichage des premières cliques détectées (n = 45)
head(Cliques_3, 3)
```

```
[[1]]
+ 3/34 vertices, named, from 4b458a1:
[1] Mr Hi   Actor 2 Actor 3

[[2]]
+ 3/34 vertices, named, from 4b458a1:
[1] Actor 29 Actor 32 John A

[[3]]
+ 3/34 vertices, named, from 4b458a1:
[1] Actor 32 Actor 33 John A
```

Pour connaître la ou les cliques au sens strict présentes dans le réseau, utilisez la fonction `largest_cliques`.

```
# Cliques "les plus grandes"
Cliques_lar <- largest_cliques(karate)

# Affichage
head(Cliques_lar)
```

```
[[1]]
+ 5/34 vertices, named, from 4b458a1:
[1] Actor 2 Mr Hi   Actor 4 Actor 3 Actor 8

[[2]]
+ 5/34 vertices, named, from 4b458a1:
[1] Actor 2  Mr Hi    Actor 4  Actor 3  Actor 14
```

4.2 Détection de communautés

Huit algorithmes de détection de communautés sont implémentés dans `igraph` :

- `cluster_edge_betweenness()`, basé sur l'intermédiarité des liens ;
- `cluster_fast_greedy()`, optimisant la modularité avec une approche gloutonne (*greedy*) ;
- `cluster_walktrap()`, basé sur des marches aléatoires ;
- `cluster_spinglass()`, basé sur la recherche de graphes hamiltoniens (graphe possédant au moins un cycle passant par tous les sommets une fois au plus) ;
- `cluster_leading_eigen()`, basé sur les vecteurs propres ;
- `cluster_label_prop()`, basé sur la diffusion entre sommets voisins ;
- `cluster_infomap()`, basé sur des marches aléatoires ;
- `cluster_louvain()`, méthode très utilisée car rapide, basée sur la maximisation locale de la modularité ;
- `cluster_optimal()`, maximisant la modularité (déconseillé pour les gros réseaux).

> **!** Important
>
> Un algorithme de partition d'un graphe se choisit en fonction de la structure du réseau : il est peu pertinent de faire de la détection de communautés sur un arbre (réseau acyclique) ou sur un réseau de type centre-périphérie. Un autre critère à prendre en compte est la taille du graphe, certains algorithmes étant inadaptés aux grands graphes. Il est conseillé de tester plusieurs algorithmes pour contrôler la stabilité des communautés détectées.

> Il existe beaucoup d'algorithmes de détection de cliques et communautés. Les différentes méthodes de partitionnement proposées par le package `igraph` sont expliquées dans ce billet de blog.

`cluster_edge_betweenness()` permet par exemple de créer une partition des sommets par déconnexion progressive du graphe, en supprimant les arêtes dans l'ordre décroissant de leur centralité d'intermédiarité.

```
karate_cluster_betweness <- cluster_edge_betweenness(karate, weights=NULL)
```

L'objet renvoyé contient les éléments suivants :

```
names(karate_cluster_betweness)
```

```
[1] "removed.edges"    "edge.betweenness" "merges"           "bridges"
[5] "modularity"       "membership"       "names"            "vcount"
[9] "algorithm"
```

Vous pouvez utiliser le `$` pour les sélectionner.

`membership` contient la classe d'appartenance de chaque sommet :

```
karate_cluster_betweness$membership
```

```
 [1] 1 1 2 1 3 3 3 1 4 2 3 1 1 2 4 4 3 1 4 1 4 1 4 5 5 5 6 5 2 6 4 4 4 4
```

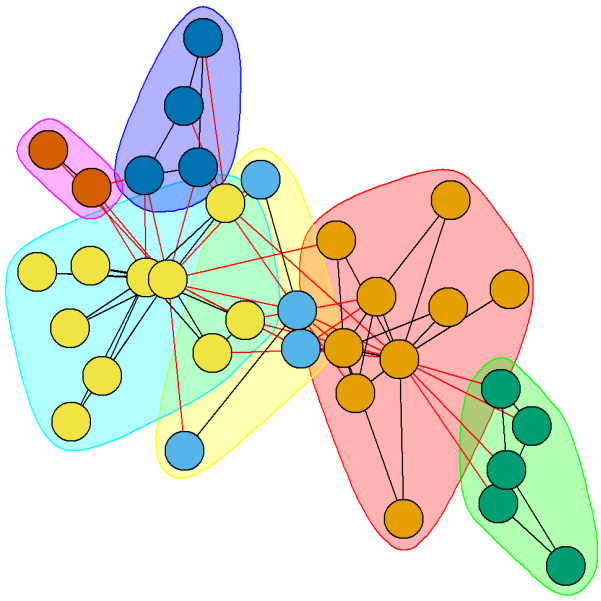
Il existe différents indicateurs permettant d'évaluer la qualité de la partition obtenue. L'un des plus fréquemment utilisés est la modularité (`modularity`). Plus cet indicateur est proche de 1, meilleure est la partition obtenue.

```
modularity(karate_cluster_betweness)
```

```
[1] 0.345299
```

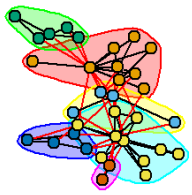
4.3 Affichage des communautés

```
plot(karate_cluster_betweness, karate, vertex.label=NA)
```

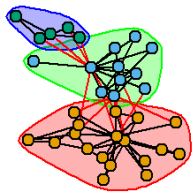


Résultats des différents algorithmes de détection de communautés proposés.

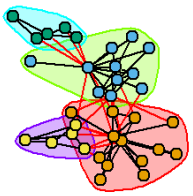
cluster_edge_betweenness
Modularité : 0.35



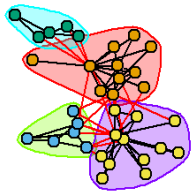
cluster_fast_greedy
Modularité : 0.43



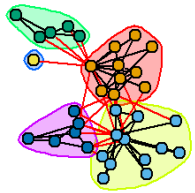
cluster_walktrap
Modularité : 0.44



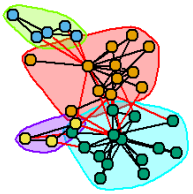
cluster_spinglass
Modularité : 0.22



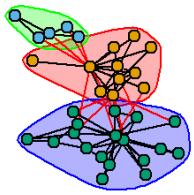
cluster_leading_eigen
Modularité : 0.44



cluster_label_prop
Modularité : 0.41



cluster_infomap
Modularité : 0.43

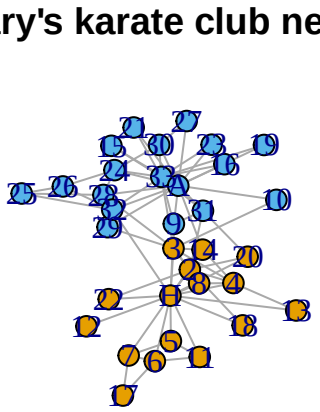


5 Représentation graphique

5.1 Afficher un objet `igraph`

Il est possible d'afficher le graphe avec la fonction `plot()` ou `plot.igraph()`.

```
plot(karate)
```

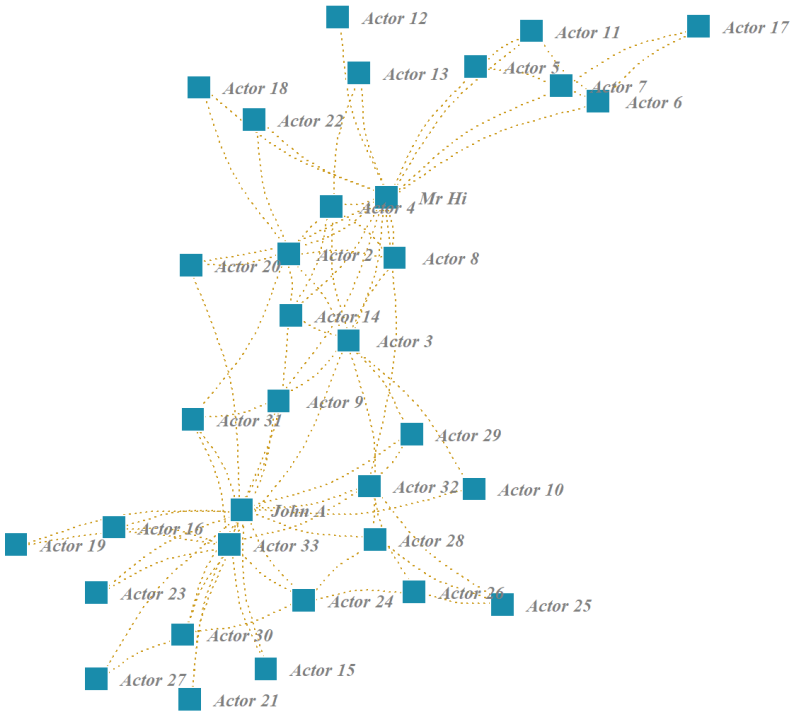


5.2 Mise en forme

Une série d'arguments permet d'améliorer la représentation du graphe. La liste des paramètres graphiques est consultable à ce lien.

Exemple de mise en forme.

```r
plot(karate,
     vertex.color = "#198cab",       # couleur des sommets
     vertex.frame.color = "white",   # couleur des contours des sommets
     vertex.shape = "square",        # forme des sommets
     vertex.size = 7,                # taille des sommets
     vertex.label = V(karate)$name, # étiquettes des sommets
     vertex.label.color = "grey50", # couleur des étiquettes des sommets
     vertex.label.font = 4,          # police des étiquettes des sommets
     vertex.label.cex = 0.8,         # taille des étiquettes des sommets
     vertex.label.dist = 2.3,        # distance du label / au centre du sommet
     vertex.label.degree = 0,        # position / au centre du sommet (0 = droite, "pi"= ga
     edge.color = "goldenrod3",      # couleur des arêtes
     edge.curved=.2,                 # courbure des arêtes (0 = droit)
     # arrow.mode = 3,               # type de flèche
     # edge.arrow.size = 12,         # taille de la pointe de la flèche
     edge.width = 1,                 # largeur des arêtes
     edge.lty = 3,                   # type de trait (1=plein, 2=pointillés, 3=points...)
     margin =  c(0,0,0.2,0),              # marges
     main = graph_attr(karate, "name"),  # titre principal
     sub = graph_attr(karate, "Author")) # sous-titre
```

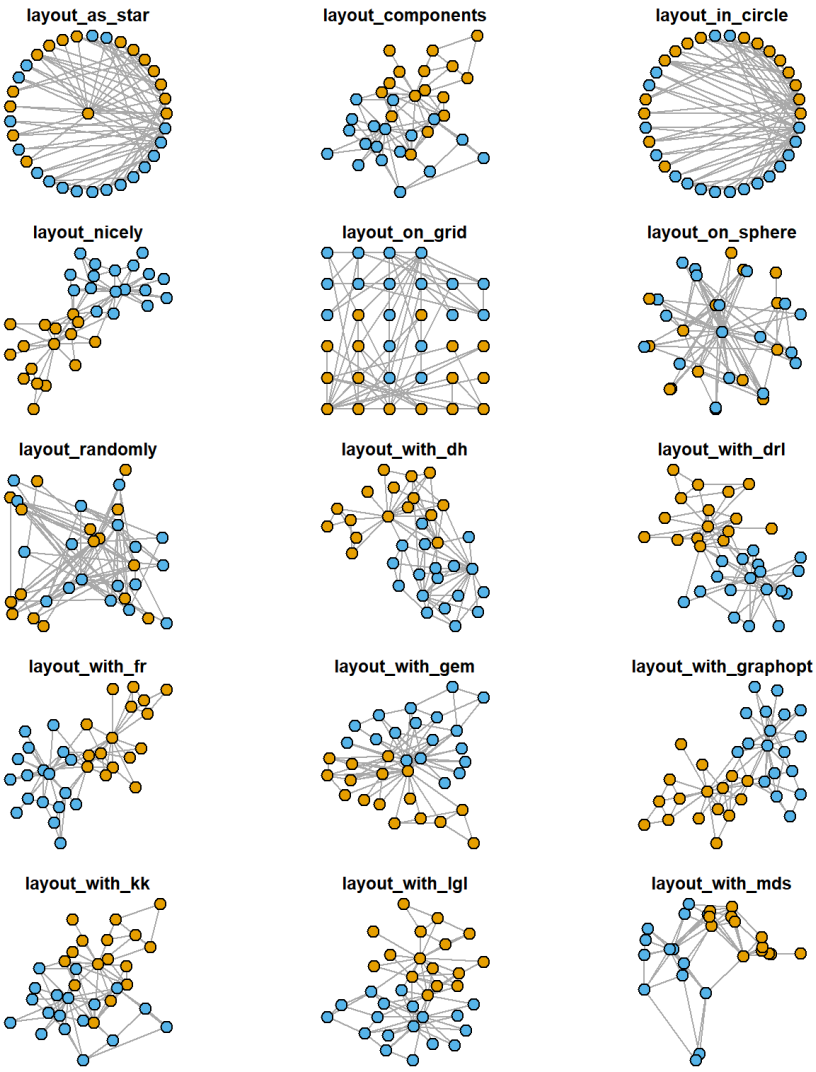


5.3 Disposition des sommets

5.3.1 Les différents `layout`

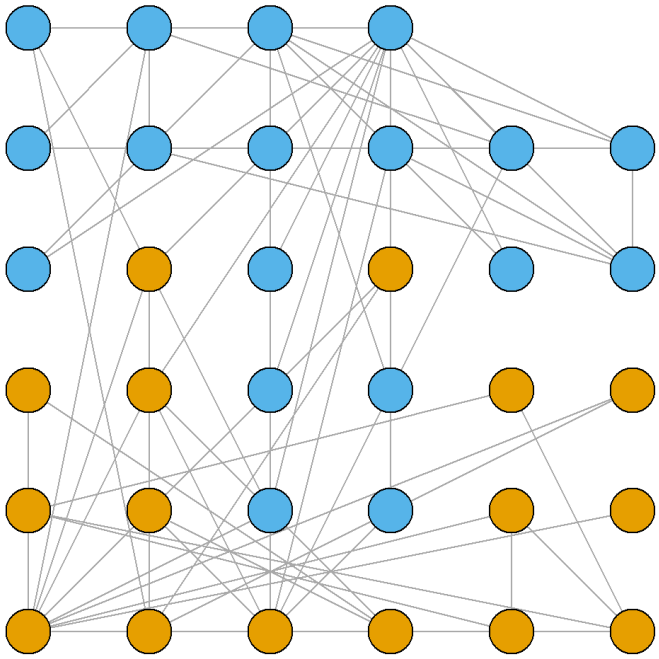
L'argument `layout` permet d'utiliser des algorithmes de placement.

Quelques exemples de spatialisation disponibles



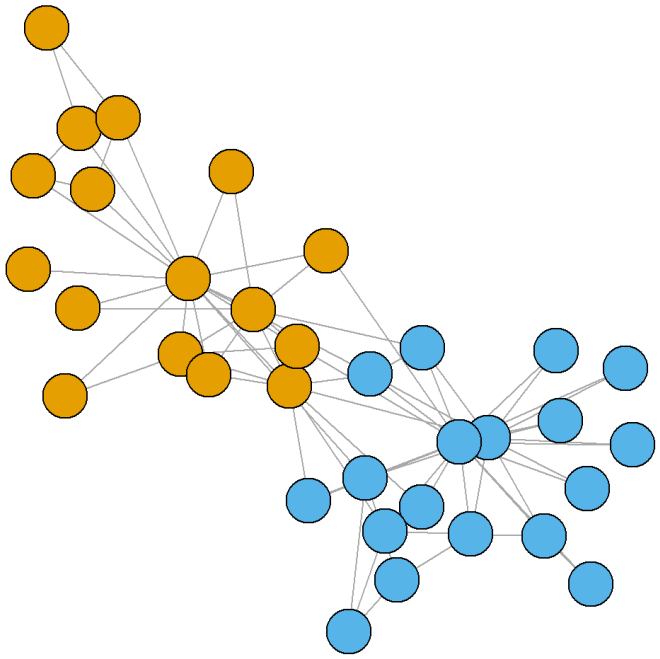
Garder le `layout` en mémoire dans un objet.

```
mise_en_page <- layout_on_grid(karate)

# Affichage
plot(karate, vertex.label=NA, layout = mise_en_page)
```



Définir le `layout` comme un attribut du graphe.

```
karate <- set_graph_attr(karate, "layout", layout_nicely(karate))

# Affichage
plot(karate, vertex.label=NA)
```



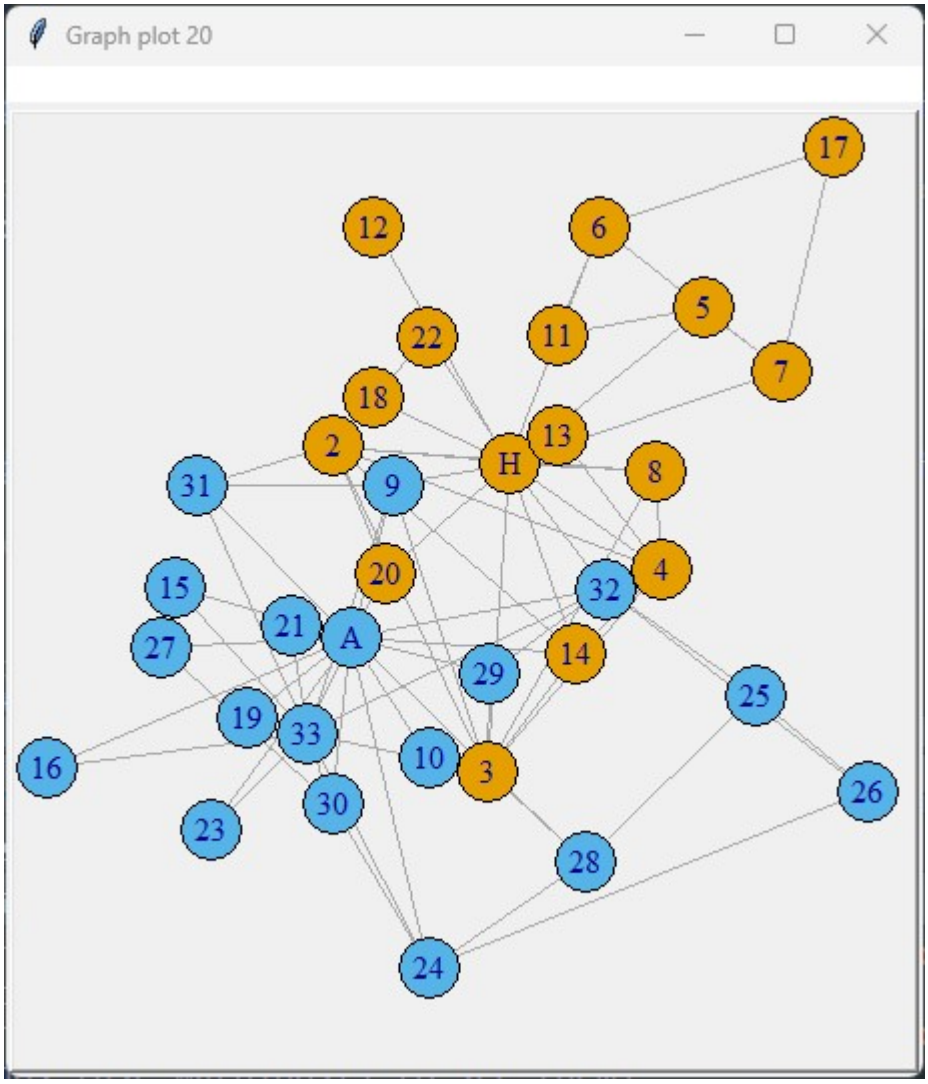
5.3.2 `layout` manuel

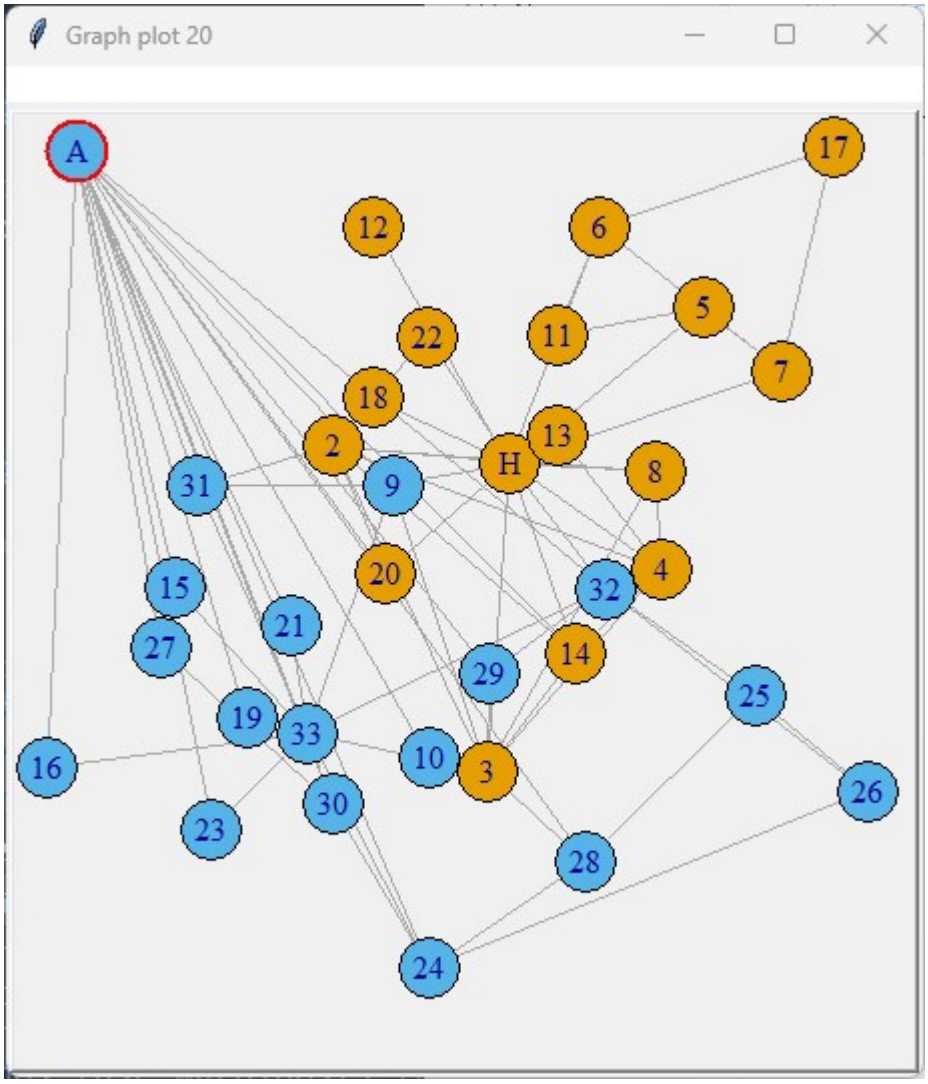
La fonction `tkplot()` permet de déplacer les sommets manuellement.

```
# Ouverture d'une fenêtre graphique dans laquelle on peut déplacer les sommets
tkid <- tkplot(karate)
```

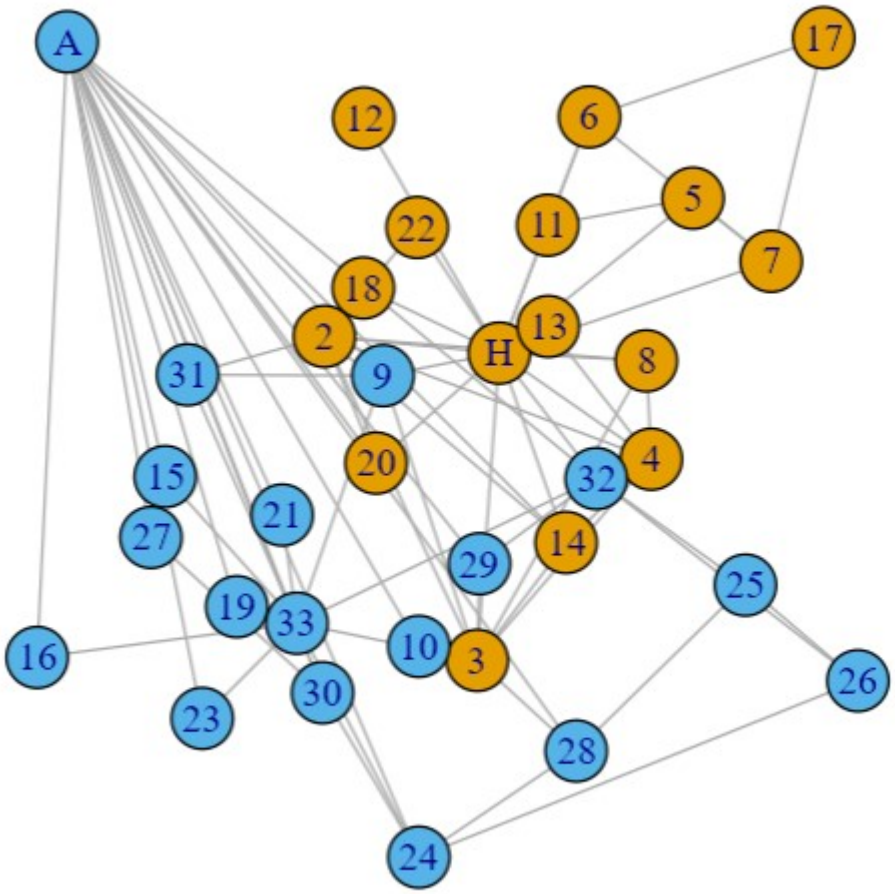
Enregistrer les nouvelles coordonnées pour les réutiliser.

```
# Enregistrement des nouvelles coordonnées des sommets
my_layout <- tkplot.getcoords(tkid)

# Fermeture de la fenêtre graphique
tk_close(tkid, window.close = T)
```


Graph plot 20
17
12
6
5
22
11
7
18
13
2
H
8
31
9
20
4
15
32
21
A
27
14
29
25
19
33
10
3
16
26
30
23
28
24


Graph plot 20
A
17
12
6
5
22
11
7
18
13
2
H
8
31
9
20
4
15
32
21
27
14
29
25
19
33
10
3
16
26
30
23
28
24

```r
# Plot avec son layout personnalisé
plot(karate, layout=my_layout)
```



5.3.3 `layout` “géographique”

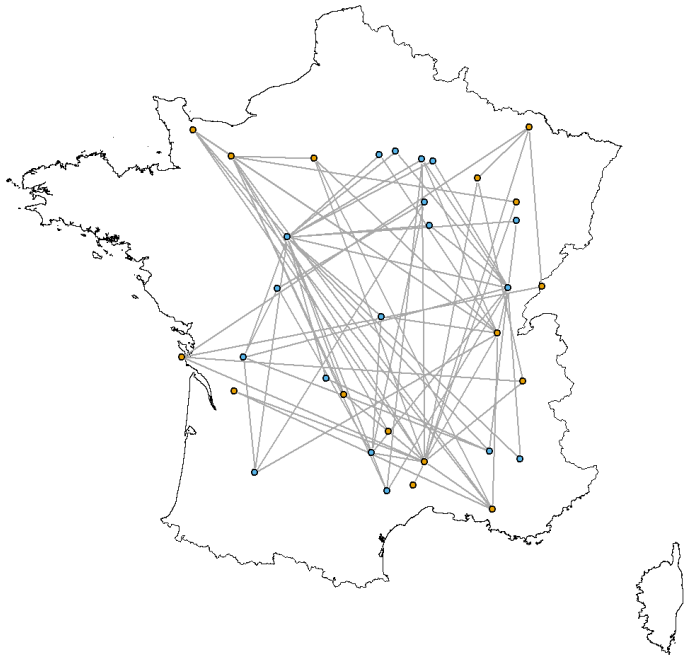
> **!** Important
>
> L’exemple présenté ci-dessous a une vocation pédagogique ; les praticiens du club étudié par Wayne Zachary ne s’entraînaient pas en France métropolitaine… Les coordonnées géographiques utilisées sont fictives.

```r
# Vecteur aléatoire de longitude
V(karate)$longitude <- runif(n=34, min=-1.5, max=6.5)

# Vecteur aléatoire de latitude
V(karate)$latitude <- runif(n=34, min=43.5, max=49.3)

# Construction d'une matrice de coordonnées
coordinates <- cbind(long = V(karate)$longitude, lat = V(karate)$latitude)

# Affichage de la matrice de coordonnées
coordinates
```

```
           long      lat
 [1,]  0.8272439 44.48787
 [2,]  6.1489467 45.44594
 [3,]  2.3949719 45.34327
 [4,]  4.7588213 47.48775
 [5,]  3.3687344 46.46219
 [6,]  2.9209604 45.09158
 [7,] -1.2167297 46.96819
 [8,]  4.5924308 47.28504
 [9,]  3.2449178 47.49970
[10,]  4.4978310 43.88823
[11,]  2.0403176 45.93439
[12,] -0.8384531 45.93220
[13,]  0.1387500 47.54216
[14,]  1.9363203 48.64917
[15,] -1.0512245 47.52344
[16,]  5.3074760 45.40351
[17,]  3.1039619 46.09548
[18,]  1.5183359 48.28779
[19,]  0.4550429 45.27698
[20,]  4.8663348 48.71502
[21,]  2.3729167 47.94086
[22,]  2.5028540 47.83773
[23,] -0.2305120 47.73770
[24,]  4.0535639 45.02845
[25,]  1.7870574 45.81958
[26,] -1.0238897 46.93732
[27,]  3.0250334 44.10580
[28,]  4.7257727 44.16134
[29,]  0.6672566 48.52689
```

```
[30,]  2.0843445 43.58570
[31,]  4.2938060 45.13998
[32,]  1.0405894 46.99387
[33,]  2.2885339 49.14941
[34,]  4.4028406 47.14554
```

```r
# Couche géographique des limites de la France métropolitaine
library(raster)
france <- getData('GADM', country='FRA', level=0)

# Affichage du fond de carte
plot(france, lwd=0.2)

# Affichage du graphe
plot(karate, layout = coordinates, rescale = FALSE,
     vertex.label = NA, main = "igraph dans l'espace", add = TRUE)
```



> **!** Important
>
> Attention, cette méthode fonctionne uniquement avec des latitudes et longitudes et pas avec des coordonnées projetées.

5.4 Les attributs graphiques

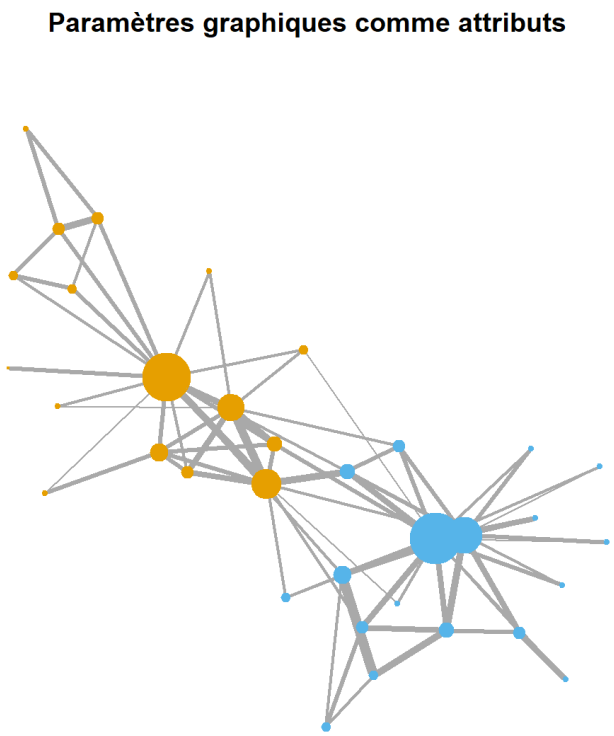
Certains paramètres graphiques peuvent être renseignés comme attributs des *vertices* et des *edges*. C'est par exemple le cas pour les attributs :

- color (*vertices* et *edges*)
- size (*vertices*)
- shape (*vertices*)

- frame.color (*vertices*)
- frame.width (*vertices*)
- label (*vertices*)
- label.color (*vertices*)
- width (*edges*)
- ...

La liste des attributs-paramètres graphiques est indiquée sur cette page.

```
# vertex.color = couleur des sommets
V(karate)$color  <- V(karate)$Faction

# vertex.frame.color = couleur de la bordure des sommets
V(karate)$frame.color <- V(karate)$Faction

# vertex.size = taille des sommets
V(karate)$size  <- degree(karate)

# edge.width = largeur des arêtes
E(karate)$width  <- E(karate)$weight

plot(karate, vertex.label = NA,
     main = "Paramètres graphiques comme attributs")
```

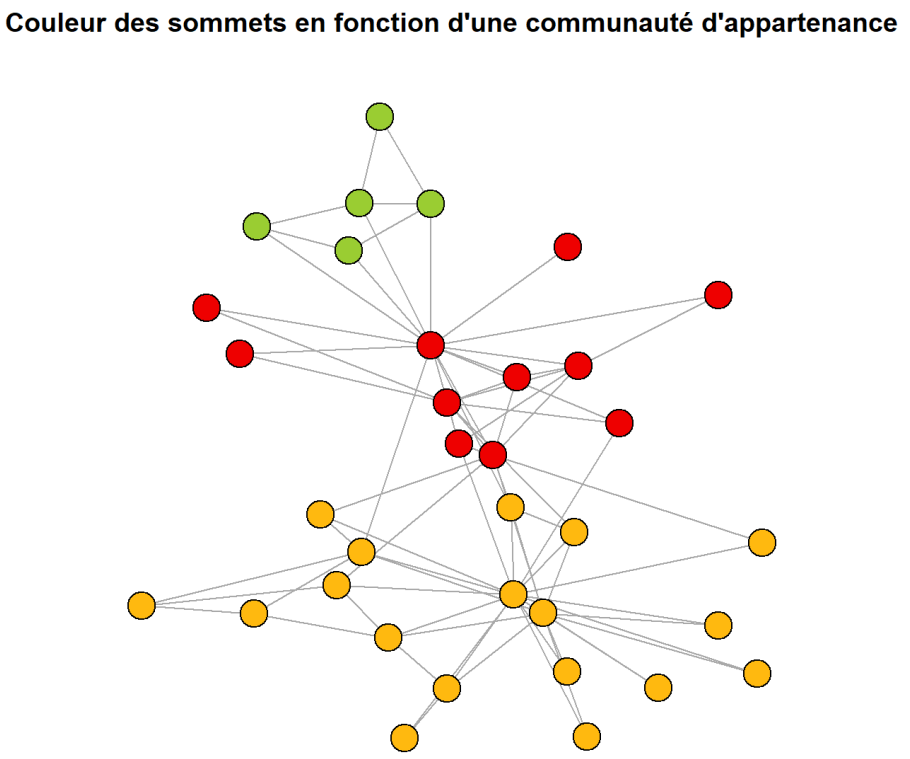


Définir une forme en fonction des modalités d'un attribut avec `ifelse`.

```
# Création d'une nouvelle variable "position"
V(karate)$position <-  c("Maître", "Élève", "Élève", "Élève", "Élève", "Élève",
                         "Élève", "Élève", "Élève",  "Élève", "Élève", "Élève",
                         "Élève", "Élève", "Élève", "Élève", "Élève", "Élève",
                         "Élève", "Élève", "Élève", "Élève", "Élève", "Élève",
                         "Élève", "Élève", "Élève", "Élève", "Élève", "Élève",
                         "Élève", "Élève", "Élève", "Maître")

# vertex.shape = forme des sommets
V(karate)$shape <- ifelse(V(karate)$position %in% "Élève", "square",
                          ifelse(V(karate)$position %in% "Maître", "circle","none"))
```

Définir une forme en fonction des modalités d'un attribut en utilisant un `factor`.

```
# Détection de communauté
# Algorithme de clustering (3 classes demandées)
com_karate <- cluster_spinglass(karate, spins=3)

# Communauté d'appartenance comme attribut des noeuds
V(karate)$membership <- com_karate$membership

# Choix de couleurs personnalisé
colrs <- c("red2","darkgoldenrod1","olivedrab3")
V(karate)$color  <- colrs[as.factor(V(karate)$membership)]
```

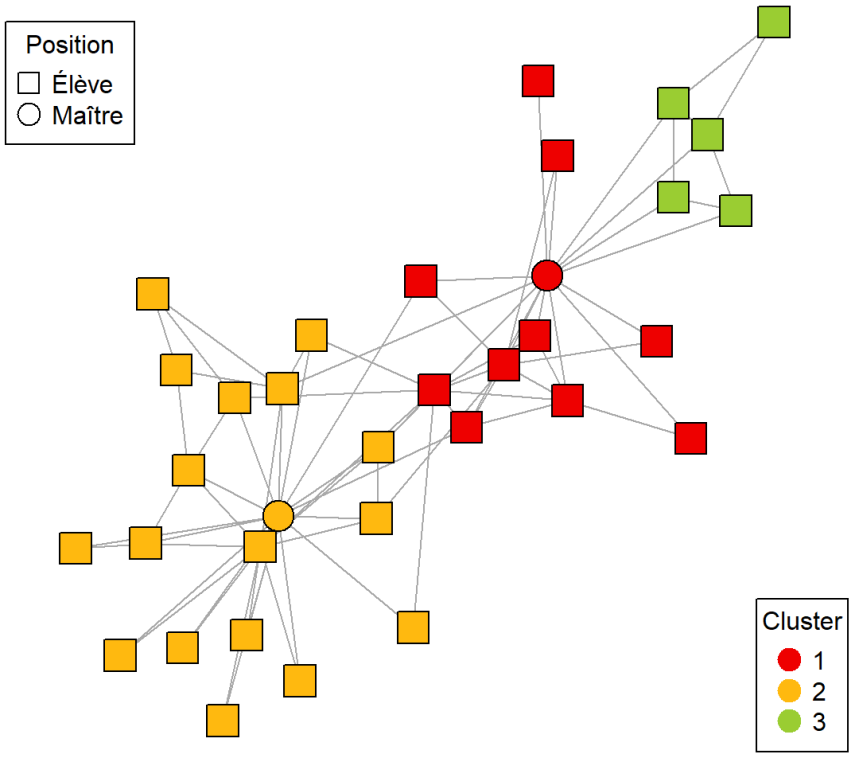


5.5 Ajouter une légende

```
plot(karate,
     vertex.label = NA,
     vertex.size = 9,
     layout = layout_nicely,
     main = "Ajouter une légende")

legend(x = -1.2, y = 1,
       legend = levels(as.factor(V(karate)$position)),
       pch = c(22, 21),
       pt.cex=2,
```

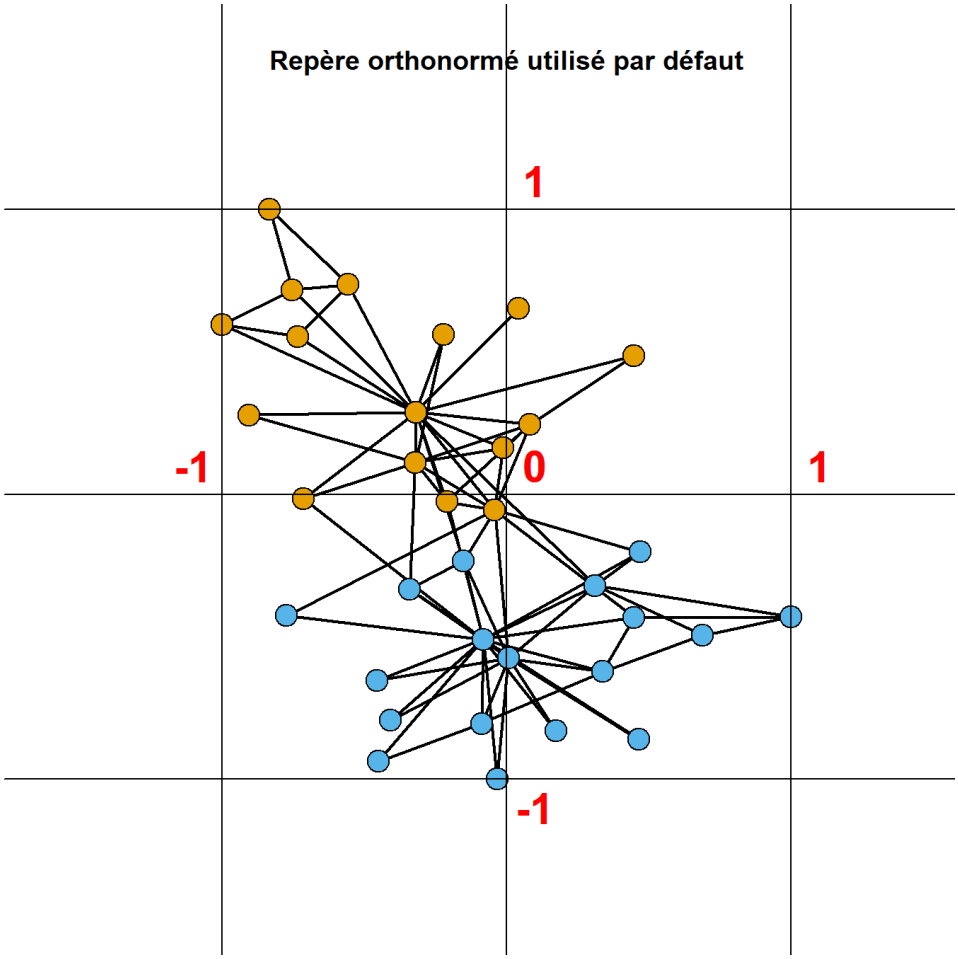
```
        bty="o",
        title = "Position")

legend(x = 0.95, y = -0.65,
        legend = levels(as.factor(V(karate)$membership)),
        pch=20,
        col= colrs,
        pt.cex=3,
        bty="o",
        title = "Cluster")
```

Ajouter une légende



> **Note**
>
> Il est actuellement compliqué de construire une légende pour les symboles proportionnels (sommets et liens), il existe néanmoins des solutions...

Les sommets d'un objet `igraph` sont par défaut représentés sur un repère orthonormé étendu de -1 à 1 sur l'axe des abscisses et des ordonnées.

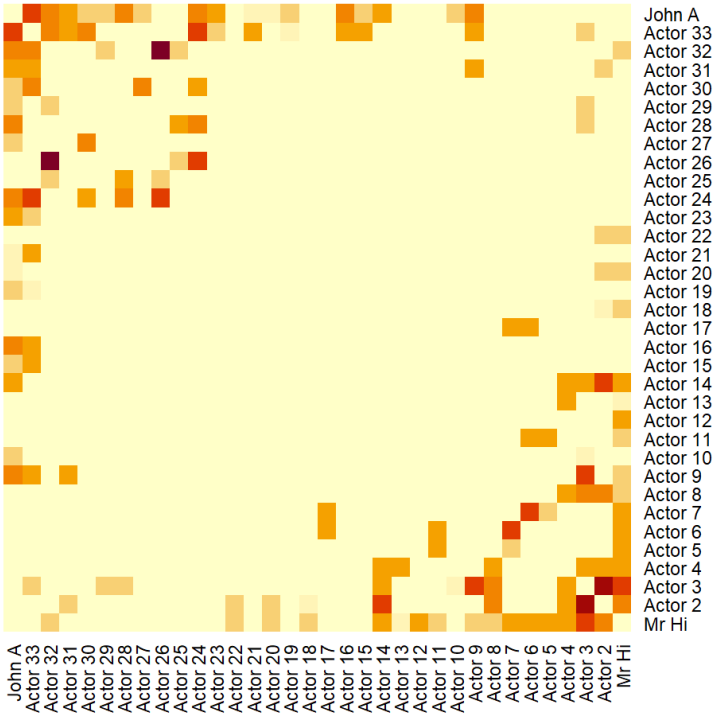


5.6 Représenter la matrice d'adjacence

Comme tous les logiciels d'analyse de réseau, `igraph` privilégie la représentation dite lien - noeud. Il est possible d'utiliser d'autres modes de représentation, comme par exemple la forme matricielle. Cependant, dans la mesure où la matrice d'adjacence n'est pas soumise à

un algorithme de réordonnancement, ce mode de représentation n'est pas nécessairement plus lisible.

```
# Matrice d'adjacence
mat <- as_adjacency_matrix(karate, attr="weight", sparse=FALSE)

# Noms de colonnes et lignes de la matrice
colnames(mat) <- V(karate)$name
rownames(mat) <- V(karate)$name

# Représentation matricielle
heatmap(mat[,34:1], Rowv = NA, Colv = NA, scale="none", margins=c(5,5))
```



Conclusion

`igraph` est un package relativement complet et plutôt simple à utiliser. Les attributs des liens et des sommets sont particulièrement faciles à gérer. Cependant, le package `igraph` présente quelques limites :

- pas d'indicateurs pour les graphes bimodaux. Pour étudier ce type de réseau, vous pouvez utiliser les packages `tnet` et `bipartite` ;
- prise en charge minimale des liens multiples (possibilité d'agréger les liens). Dans ce cas, le package `multinet` apporte des solutions (cf https://groupefmr.hypotheses.org/6016) ;
- pas d'indicateurs spécifiques aux réseaux personnels. Pour cela, utiliser le package `egor`.

Enfin, certaines méthodes d'analyse ne sont pas proposées comme le *blockmodel* ou les modèles ERGM (*Exponential Random Graph Model*). Pour ces deux types de traitements, il faut se tourner vers l'autre grand package généraliste d'analyse de réseau disponible dans R, `statnet`.

Exercice

Données

Les données et le corrigé de cet exercice sont mis à disposition *via* un projet RStudio :

Télécharger le projet Rstudio

Décompressez le dossier. Deux fichiers .csv sont à votre disposition dans le sous-répertoire *data* :

- data/**liens_d13.csv**, table de liens ;
- data/**som13.csv**, table de sommets.

Il s'agit de données de mobilités scolaires 2017 entre communes des Bouches du Rhône (réseau unimodal, orienté, valué et pouvant être cartographié pour les plus géographes d'entre nous...). La table **som13** contient des attributs socio-démographiques de l'Insee. Le dictionnaire des variables est disponible ici : https://www.insee.fr/fr/statistiques/2521169#dictionnaire.

Source : https://www.insee.fr/fr/statistiques/2521169#consulter

Questions

Les points d'interrogation en début de ligne indiquent une fonction manquante. Les solutions se trouvent :

- dans le support ;
- dans la documentation du package ;
- dans le fichier exercice_corrige.R.

1. Charger la bibliothèque

- `install.packages()`
- `library()`

```
# install.packages("igraph")
library(igraph)
```

2. Importer le tableau des sommets et des liens

- `read.csv()`
- `read.csv2()`

```
# Import des liens
lien <- read.csv2("data/liens_d13.csv", header = TRUE)

# Import de la table des sommets
som <- read.csv("data/som13.csv", sep = " ", header = TRUE)
```

3. Construire un objet `igraph` à partir de ces tableaux

- `graph_from_data_frame()`

```
# Transformation en objet igraph
g13 <- graph_from_data_frame(lien, som, directed = TRUE)

# Examen de l'objet
g13

# Que signifie DNW ?
# Directed, Named, Weighted
```

4. Visualiser le graphe

```
# Visualisation par défaut
plot(g13)
```

5. Ordre et taille du graphe

- `vcount()`
- `ecount()`

```
# Ordre (nombre de sommets)
vcount(g13)

# Taille (nombre de liens)
ecount(g13)
```

6. Supprimer les sommets isolés

```
# Suppression des isolés
g13 <- delete.vertices(g13, degree(g13) < 1)

# Visualisation
plot(g13,
     edge.arrow.size = 0.5,
     vertex.size = 5,
     vertex.label = NA)
```

7. Explorer les composantes

```
# Décompose composantes connexes
c13 <- decompose(g13)

summary(c13)

# Extraction et analyse de la plus grande composante
g <- c13[[1]]
```

```r
plot(g,
     vertex.label = NA,
     edge.arrow.size = 0.2,
     vertex.color = "red",
     vertex.size = degree(g, mode = c("all")))
```

8. Fixer des paramètres

```r
# Couleur des sommets
V(g)$color <- "red"

# Taille des sommets
V(g)$size <- 5
```

9. Calculer des indicateurs et mesurer leur relation

- Diamètre du graphe
- Distribution de l'intensité des liens
- Proportion des liens mutuels et asymétriques

```r
# Diamètre du réseau
diameter(g)

# Diamètre topologique
diameter(g,  weights = NA)

# Distribution de l'intensité des liens
summary(E(g)$weight)

# Ne conserver que les liens > x
# g <- delete.edges(g, which (E(g)$weight < x))

# Proportion de liens mutuels et asymétriques
dyad.census(g)
```

10. Relations entre centralités

Calcul des différents indicateurs de centralité.

```
# degré entrant pondéré
degin_w <- strength(g, mode = c("in"))

# degré sortant pondéré
degou_w  <- strength(g, mode = c("out"))

# degré entrant
degin <- degree(g, mode = c("in")

# degré sortant
degou <- degree(g, mode =c("out"))

# degré total
deg <- degree(g, mode = c("all"))

# intermédiarité
bet <-betweenness(g)

# proximité
clo <- closeness(g, mode= c("all"))
```

Les communes qui émettent le plus d'élèves sont-elles celles qui en reçoivent le plus ?

```
plot(degou_w, degin_w)
```

Et si on raisonne en nombre de communes et non en volume d'enfants ?

```
plot(degou, degin)
```

Les communes ayant le degré le plus élevé ont-elles aussi une forte intermédiarité ? une closeness faible ?

```
plot(deg, bet)
plot(deg, clo)
```

11. Rechercher des cliques

```
# Transformation en réseau non orienté
gno <- as.undirected(g)

# Calcul de la plus grande clique
lcl <- largest.cliques(gno)

# Attribuer une couleur unique à tous les sommets
vcol <- rep("red", vcount(gno))

# Attribuer une autre couleur aux sommets de la plus large clique
vcol[unlist(lcl)] <- "gold"

# Visualisation
plot(gno,
     vertex.label=NA,
     vertex.color=vcol)


# Lors de la transformation, on peut ne garder que certains liens pour ne garder
# que les liens mutuels
# gmut <- as.undirected(g, mode = c("mutual"))
```

12. Extraire l'ego-network du premier sommet

```
# Calcul de ego Network
egoN <- make_ego_graph(g, order = 1, mode = c("all"))

# Ego network du premier sommet (Aix-en-Provence)
egoN[[1]]
# V(egoN[[1]])$vertex.label <- V(egoN[[1]])$name

plot(egoN[[1]],
     # vertex.label = NA,
     vertex.color = "red",
     edge.arrow.size = 0.1)
```

Bonus 1 : Manipuler un réseau avec boucles et liens multiples

Construction d'un réseau avec boucles et liens multiples.

```
gbm <- d1 <- rbind(c("A","A"),
                   c("A","B"),
                   c("B","C"),
                   c("B","C"),
                   c("B","C"),
                   c("C","D"),
                   c("C","D"))
```

```
g1 <- graph.data.frame(d1, directed = FALSE)

# Examen de l'objet
g1

# Tests
is.loop(g1)
is.multiple(g1)

# Visualisation par défaut
plot(g1)

# Centralité de degré
degree(g1)

# Par défaut, simplify supprimer boucles et liens multiples
g1_2 <- simplify(g1)
plot(g1_2)

# Si je veux garder les liens multiples
g1_3 <- simplify(g1, remove.multiple = FALSE)
plot(g1_3)

# Si je veux garder une trace des boucles et des liens multiples
g1_4 <- simplify_and_colorize(g1)
g1_4

V(g1_4)$color # il y avait une boucle au premier sommet
E(g1_4)$color # il y avait 1 lien, 3 liens, 2 liens
```

Bonus 2 : Manipuler un réseau bimodal

Construction d'un réseau bimodal.

```
som <- c("a1","a2","a3","a4","A1","A2","A3")
somtype <- c(FALSE, TRUE, FALSE, TRUE, TRUE, FALSE, FALSE)
gb <- make_graph(c("a1","A1","a2","A1",
                   "a2","A2","a2","A3",
                   "a3","A2","a4","A3",
                   "a4", "A2"), directed = FALSE)


# Projection réseau biparti
V(gb)$type <- somtype

# Visualisation
plot(gb, layout = layout_as_bipartite(test))
```

```
V(gb)$color <- "yellow"
V(gb)$size <- 30
E(gb)$color <- "black"

# Projection réseau bimodal
pro <- bipartite.projection(gb)
E(pro[[1]])$weight
E(pro[[2]])$weight

par(mfrow=c(1,3))
plot(gb, layout = layout_as_bipartite(gb))
plot(pro[[1]], edge.width= E(pro[[1]])$weight)
plot(pro[[2]], edge.width= E(pro[[2]])$weight)
```

Références

Analyse de réseau avec R

Gabor Csardi, nd, *Practical statistical network analysis (with R and igraph)*

Katherine Ognyanova, 2016, *Network Analysis and Visualization with R and igraph*

Katherine Ognyanova, 2017, *Visualisation de réseaux avec R* (traduit par LB)

Jesse Sadle, 2017, *Introduction to Network Analysis with R*

Sébastien Plutniak, 2018, *L'analyse de graphes avec R : un aperçu avec igraph*

Analyse de réseau en SHS

François Briatte, depuis 2016, *An awesome list of network analysis resources*

Laurent Beauguitte, 2023, *L'analyse de réseau en sciences sociales. Petit guide pratique*