Graph Mining

Laurent Rouvière

2021-01-08

Table des matières

Présentation

Ce tutoriel présente une introduction au **graph mining** avec **R**. On pourra trouver :

- les supports de cours associés à ce tutoriel ainsi que les données utilisées à l'adresse suivante https://lrouviere.github.io/INP-HB/ ;
- le tutoriel sans les correction à l'url https://lrouviere.github.io/TUTO_GRAPHES/
- le tutoriel avec les corrigés (à certains moment) à l'url https://lrouviere.github.io/TUTO_GRAPHES/correction/.

Il est recommandé d'utiliser **mozilla firefox** pour lire le tutoriel.

Les thèmes suivants sont abordés :

- Manipulation de graphes avec **igraph** : visualisation statique et dynamique
- **Modèles** basiques pour des graphes, notamment Erdos-Renyi et SBM
- **Détection de communautés**, maximisation de la modularité
- **Clustering spectral**, pour des graphes mais aussi des "données standards".

1 Manipulation de graphes avec igraph

Le but de ce tutoriel est de se familiariser avec les principales fonctions du package **igraph**. On trouvera un descriptif de ce package à l'adresse http://igraph.org/r/. On pourra également consulter le tutoriel (très complet) suivant : http://kateto.net/networks-r-igraph.

Nous commençons par charger le package

```
library(igraph)
```

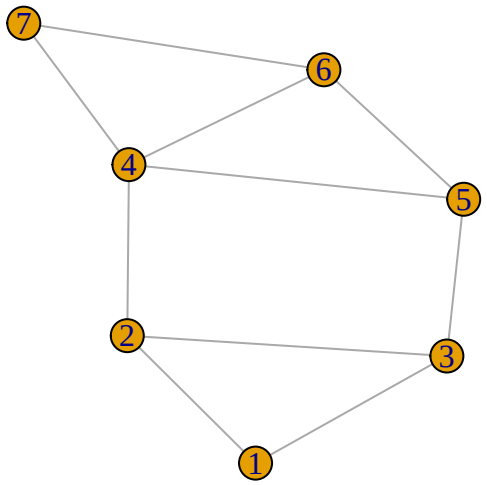
1.1 Construction de graphes avec igraph

Comme pour des données classiques, il est possible de construire des graphes directement dans **R** ou de les importer à partir de fichiers externes.

1.1.1 Quelques fonctions R pour construire des graphes

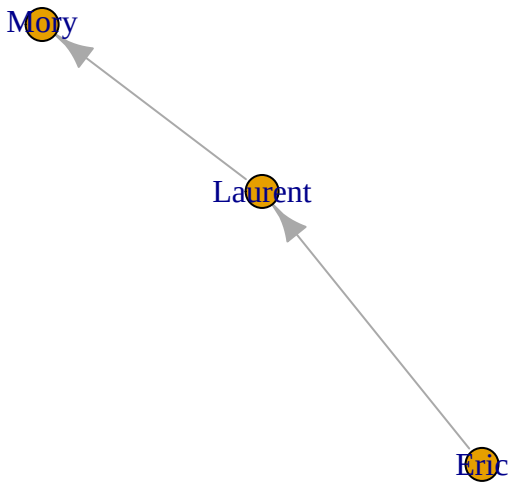
L'approcher la plus naturelle est de définir un graphe à partir d'une liste d'arêtes :

```
g1 <- graph(edges=c(1,2,1,3,2,3,3,5,2,4,4,5,5,6,4,6,4,7,6,7),n=7,directed=F)
plot(g1)
```



Si la liste d'arêtes est donnée sous forme de noms, il n'est pas nécessaire d'indiquer le nombre de nœuds.

```
g2 <- graph(edges=c("Eric","Laurent","Laurent","Mory"))
plot(g2)
```

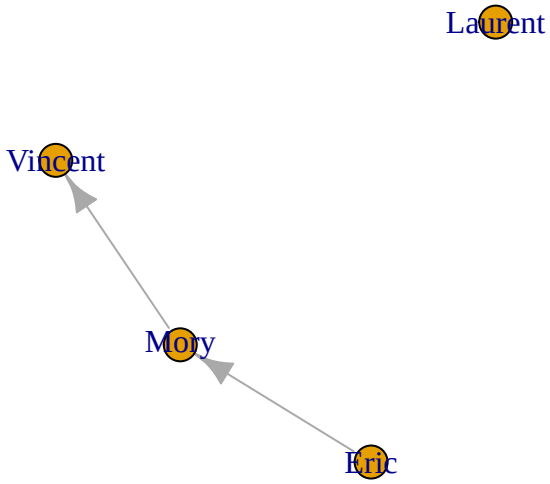


Différentes options sont proposées dans la fonction **graph** :

- `n` : le nombre maximal de nœuds
- `isolates` : ajout de nœuds isolés
- `directed` : graphes dirigés ou non
- ...

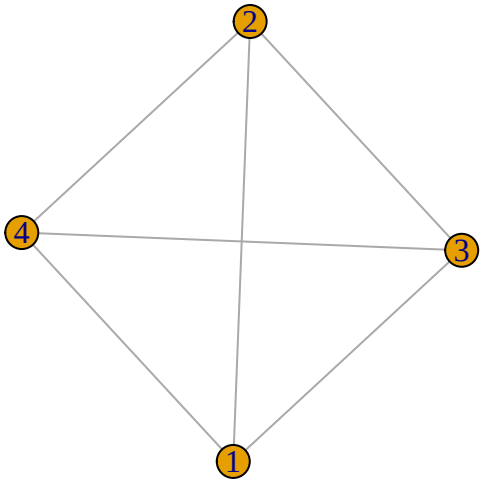
Par exemple

```
g3 <- graph(edges=c("Eric","Mory","Mory","Vincent"),isolates="Laurent")
plot(g3)
```

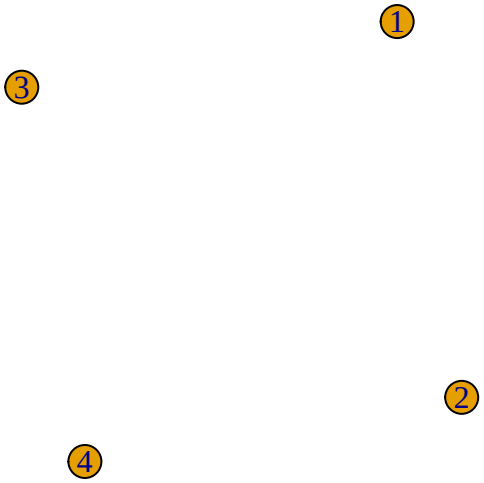


Il existe également des fonctions spécifiques qui peuvent aider à la construction de graphes, par exemple **make_full_graph** :

```
plot(make_full_graph(4))
```

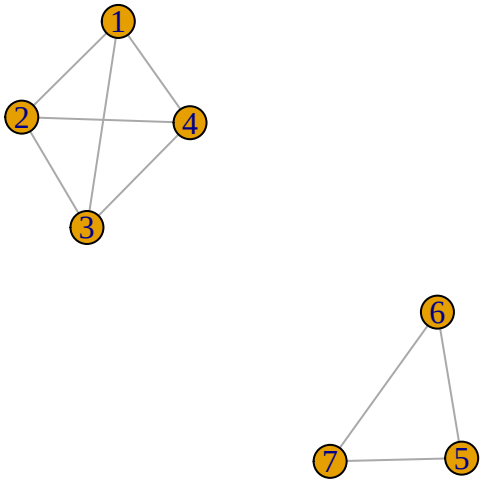



```
plot(make_empty_graph(4))
```



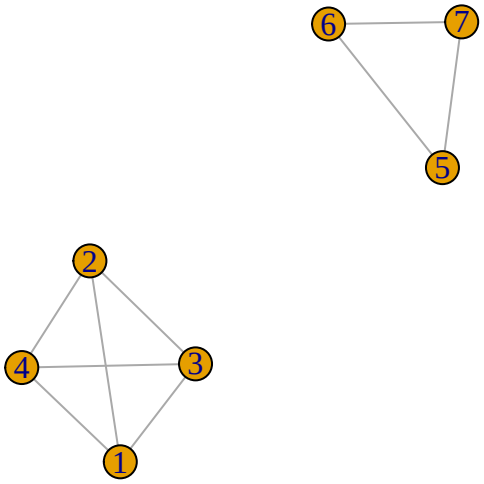
Il est également possible “d’additionner” des graphes

```
plot(make_full_graph(4)+make_full_graph(3))
```



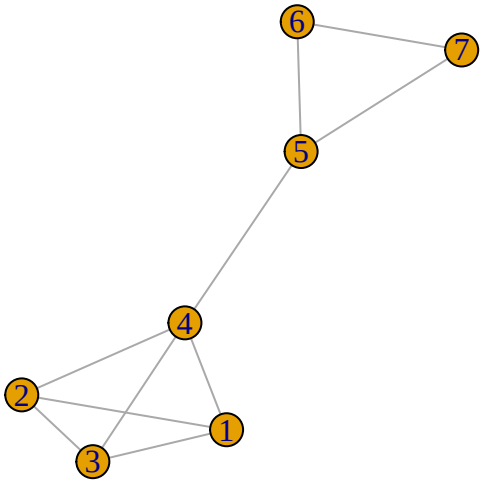
L’opérateur **pipe** permet une lecture du code plus lisible

```
(make_full_graph(4)+make_full_graph(3)) %>% plot()
```



Il est également facile d'ajouter des arêtes avec **add_edges** :

```
(make_full_graph(4)+make_full_graph(3)) %>% add_edges(c(4,5)) %>% plot()
```



Exercice 1.1 (Quelques graphes spécifiques)**.**

Tester et expliquer les fonctions **make_empty_graph, make_ring** et **make_star**.

1.1.2 Construction à partir d'un fichier externe

Le plus souvent, on aura à récupérer des données récoltées dans des fichiers `txt` ou `csv` pour construire le graphe.

Exercice 1.2 (Importation)**.**

On considère le jeu de données **Friendship-network_data_2013.csv** qui se trouve sur le site http://www.sociopatterns.org/datasets/high-school-contact-and-friendship-networks/. Ces données concernent des relations entre étudiants.

1. Importer les données à l'aide de **read.table**.
2. Ce fichier contient 2 colonnes et chaque colonne contient une arête. Visualiser le graphe. On pourra utiliser **graph_from_data_frame**.
3. On considère un graphe permettant de visualiser des connexions entre médias :

— les nœuds sont définis dans le fichier `Dataset1-Media-Example-NODES.csv`
— les arêtes dans le fichier `Dataset1-Media-Example-EDGES.csv`.

Importer ces fichiers.

4. Construire le graphe `igraph` associé à ces deux fichiers à l'aide de **graph_from_data_frame**.
5. La fonction **read_graph** peut s'adapter à de nombreux formats de graphe :

```
read_graph(file, format = c("edgelist", "pajek", "ncol", "lgl", "graphml", "dimacs", "graphdb", "gm
```

On considère par exemple le fichier **lesmis.gml** disponible ici. Les nœuds correspondent aux personnages du roman et une arête est présente si deux personnages apparaissent dans le même chapitre. Le poids de l'arête est déterminé par le nombre de chapitres où les deux personnages sont présents. Importer le graphe à l'aide de **read_graph** et visualiser le.

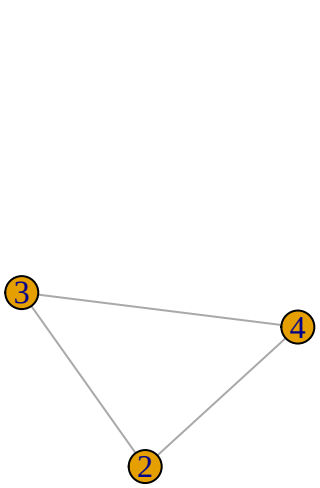
1.1.3 Matrice d'adjacence

Enfin un graphe peut également s'identifier avec une **matrice d'adjacence** :

```
A <- matrix(c(0,0,0,0,0,0,1,1,0,1,0,1,0,1,1,0), 4, 4)
A
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    0
[2,]    0    0    1    1
[3,]    0    1    0    1
[4,]    0    1    1    0
```

On pourra utiliser dans ce cas la fonction **graph_from_adjacency_matrix** pour convertir la matrice en un objet `igraph` :

```
G <- graph_from_adjacency_matrix(A, mode='undirected')
plot(G)
```



On peut bien entendu faire l'opération inverse et calculer la **matrice d'ajacence** d'un graphe :

```
as_adj(G)
4 x 4 sparse Matrix of class "dgCMatrix"

[1,] . . . .
[2,] . . 1 1
[3,] . 1 . 1
[4,] . 1 1 .
```

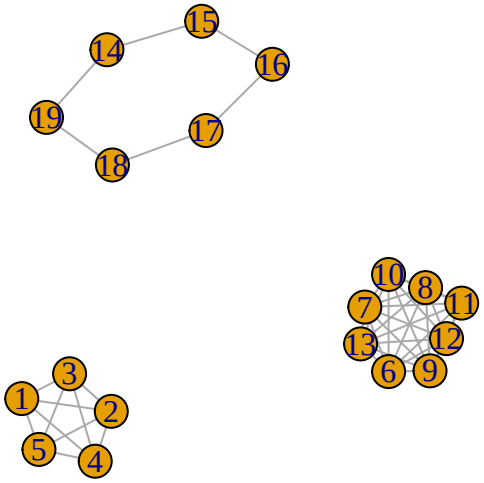
1.2 Visualisation d'un graphe

Un des intérêts principaux du graph mining et de visualiser les connexions entre les nœuds à l'aide d'un graphe. Se pose bien entendu la question (difficile) de la position des nœuds dans le plan pour obtenir la visualisation la plus pertinente du graphe. On peut ensuite s'interroger sur des outils classiques qui vont permettre de colorier les nœuds, d'utiliser différents symboles pour les arêtes, etc...

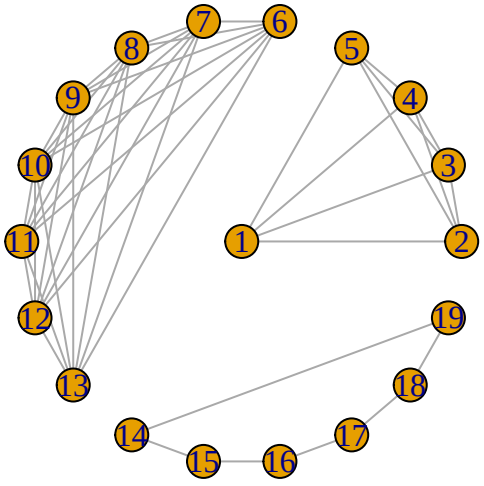
1.2.1 Network layouts : algorithmes usuels de visualisation

Un graphe peut être visualisé à l'aide de plusieurs algorithmes. On pourra trouver un descriptif ici. On se contentera de donner différents layouts pour l'exemple suivant :

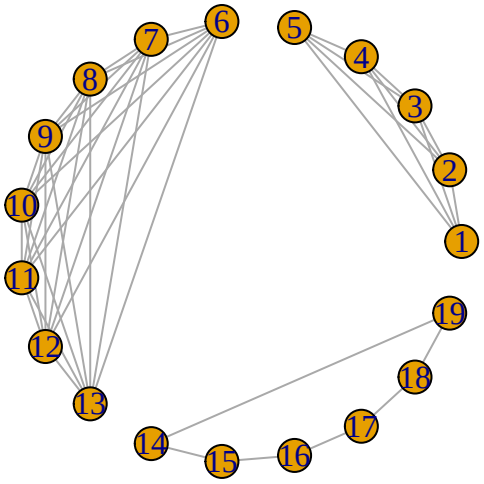
```
G <- make_full_graph(5)+make_full_graph(8)+make_ring(6)
plot(G)
```



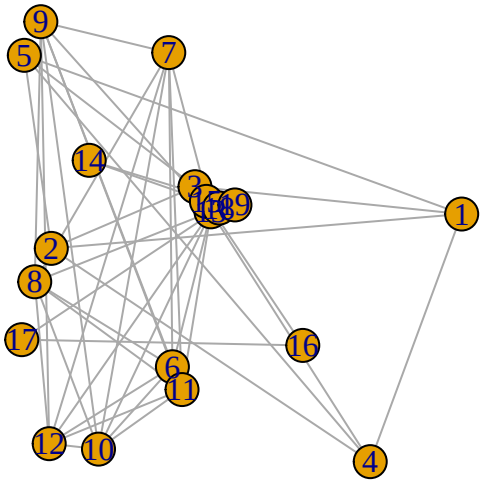

```
plot(G,layout=layout_as_star(G))
```




```
plot(G,layout=layout.circle(G))
```

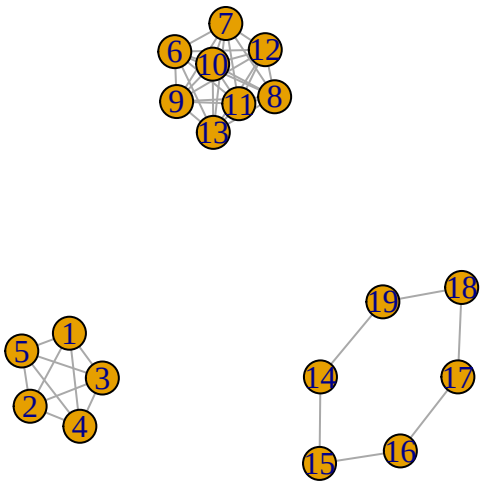



```
plot(G,layout=layout_randomly(G))
```

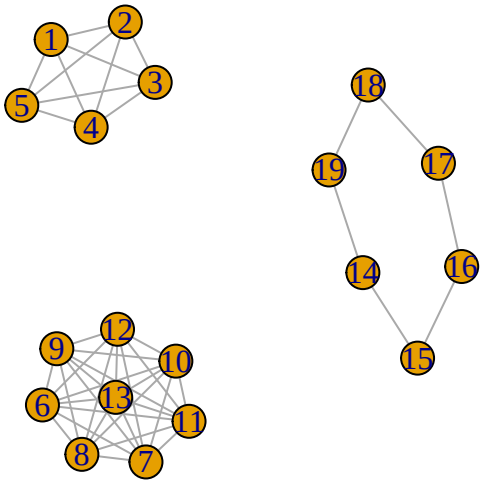


Deux algorithmes sont connus pour avoir des visualisations jugées “esthétiques”. L’idée, très rapidement, est d’essayer d’obtenir la position des nœuds et des arêtes de façon uniforme dans le plan. Pour plus d’informations sur ce sujet difficile on pourra consulter cet article.

```
plot(G,layout=layout_with_fr(G))
```



```
plot(G,layout=layout_with_kk(G))
```



Enfin la fonction **tktplot**

```
tkplot(G)
```

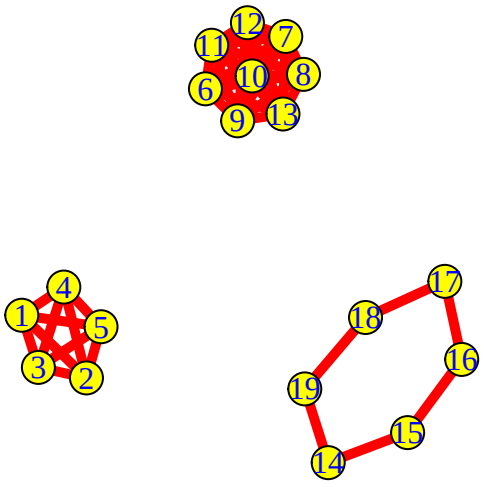
1.2.2 Personnalisation du graphe

Il est bien entendu possible de modifier les couleurs, tailles... des nœuds et arêtes. Deux stratégies sont possibles avec **igraph** :

- utiliser les options de **plot.igraph** : vertex.color, vertex.shape, vertex.size, vertex.label... et edge.color, edge.label, edge.width...
- travailler sur les nœuds et arêtes séparément à l'aide des fonctions **V()** et **E()**.

La fonction **plot.igraph** :

```
plot(G,
     vertex.color="yellow",vertex.size=15,vertex.label.color="blue",
     edge.color="red",edge.width=5)
```



On peut également modifier les paramètre des nœuds

```
G1 <- G
V(G1)$color <- "red"
V(G1)$size <- 15
```

et des arêtes

```
E(G1)$color <- "blue"
E(G1)$size <- 3
```

On a ainsi

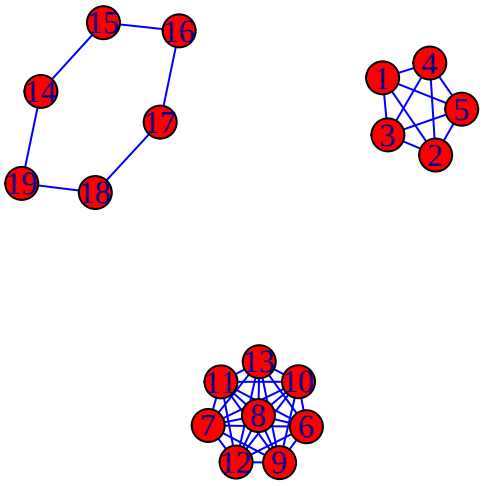
```
vertex_attr(G1)
$color
 [1] "red" "red" "red" "red" "red" "red" "red" "red" "red"
[10] "red" "red" "red" "red" "red" "red" "red" "red" "red"
[19] "red"

$size
 [1] 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15 15
[19] 15
edge_attr(G1)
$color
 [1] "blue" "blue" "blue" "blue" "blue" "blue" "blue" "blue"
 [9] "blue" "blue" "blue" "blue" "blue" "blue" "blue" "blue"
[17] "blue" "blue" "blue" "blue" "blue" "blue" "blue" "blue"
[25] "blue" "blue" "blue" "blue" "blue" "blue" "blue" "blue"
[33] "blue" "blue" "blue" "blue" "blue" "blue" "blue" "blue"
[41] "blue" "blue" "blue" "blue"

$size
 [1] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
[29] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
```

et on peut visualiser le graphe (sans option dans **plot.igraph**) :

```
plot(G1)
```



Exercice 1.3 (Gérer les couleurs avec igraph)**.**

1. Construire un graphe avec 3 composantes composantes connexes de taille 10, 14 et 8.
2. Colorier les nœuds de chaque composantes d'une couleur différente.

3. Relier les composantes en ajoutant 2 arêtes (et pas plus).
4. Utiliser une couleur et une taille différente pour les deux arêtes crées.

Exercice 1.4 (Customiser un graphe)**.**

On considère le graphe **net** sur les médias défini dans l'exercice 1.2. Représenter le graphe en ajoutant :

— le nom des nœuds (**media**)
— une couleur différente en fonction du type de média
— une taille de nœud différente en fonction de l'audience
— une taille d'arête différente en fonction du poids (`weight`)
— une couleur d'arête différente en fonction du type (`type`).

1.3 Statistiques descriptives sur les graphes

Comme pour tout les types de données, il est souvent crucial de calculer des indicateurs descriptifs sur les graphes. Nous présentons les indicateurs standards tels que le diamètre, la densité, les degrés de centralité et d'intermédiarité...

Exercice 1.5 (Quelques descripteurs)**.**

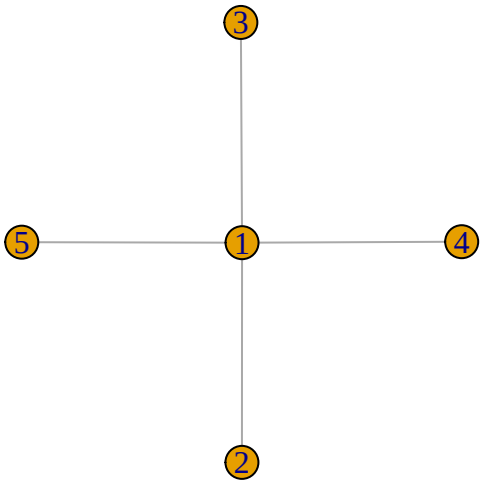
On considère toujours le graphe des exercices précédents sur les média (`net`).

1. Calculer les nombre de nœuds, d'arêtes, le diamètre et la densité du graphe.
2. Combien y a t-il de triangles dans le graphes ? On pourra calculer ce nombre de plusieurs façons.
3. Calculer la transitivité.
4. Quels sont les nœuds connectés avec le nœud 3 ? On pourra utiliser **neighbors**.
5. Étudier les composantes connexes du graphe.
6. Calculer les degrés des nœuds et représenter les avec un barplot. On pourra utiliser **degree** puis **degree_distribution**.
7. Calculer les degrés de **proximité** et d'**intermédiarité** et ordonner les observations en fonction de ces degrés.

Exercice 1.6 (Centralité et intermédiarité)**.**

On considère le graphe suivant.

```
G <- make_star(5,mode="undirected")
plot(G)
```



Calculer en utilisant la définition les degrés de centralité et d'intermédiarité des nœuds de G. Retrouver ces valeurs à l'aide de fonctions **R**.

Exercice 1.7 (Comparaison de nœuds)**.**

On considère le graphe **karate** que l'on peut récupérer dans le package **igraphdata** :

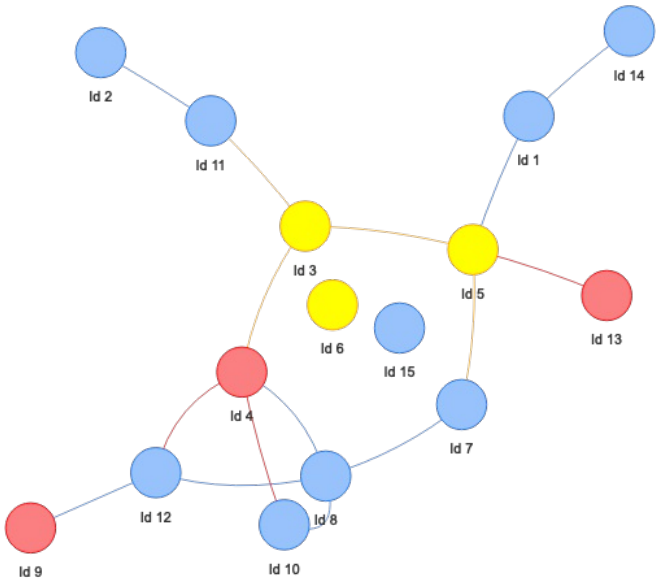
En étudiant les différents critères d'importance des nœuds, identifier les nœuds importants et interpréter.

1.4 Autres packages pour visualiser les graphes

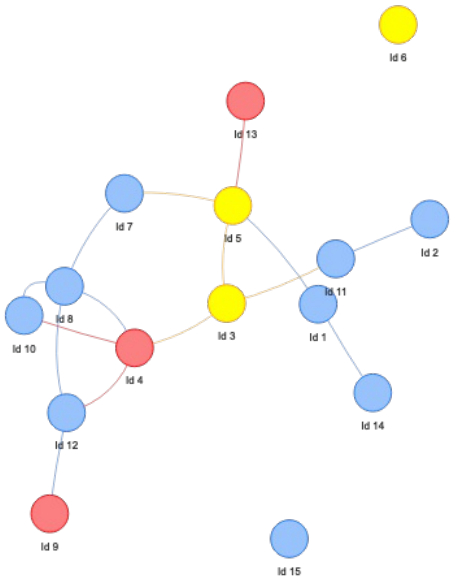
1.4.1 Graphes dynamiques avec visNetwork

Nous avons vu que le package igraph propose une visualisation statique d'un réseau. Pour donner un caractère dynamique à ce type de représentation, on pourra utiliser le package visNetwork. Une représentation standard **visNetwork** s'effectue en spécifiant les nœuds et connexions d'un graphe. Voici quelques exemples d'utilisation.

```
set.seed(1234)
nodes <- data.frame(id = 1:15, label = paste("Id", 1:15),
                    group=sample(LETTERS[1:3], 15, replace = TRUE))
edges <- data.frame(from = trunc(runif(15)*(15-1))+1,to = trunc(runif(15)*(15-1))+1)
library(visNetwork)
visNetwork(nodes,edges)
```

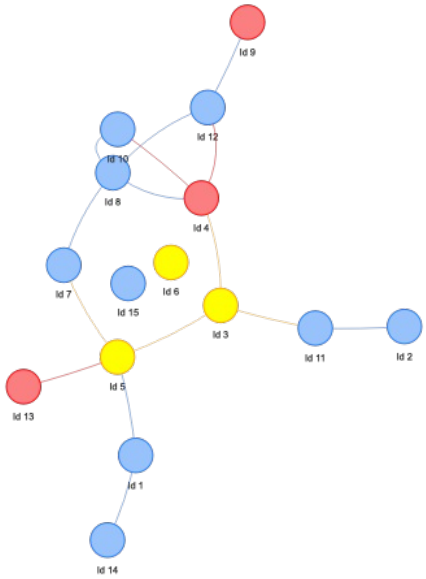



```
visNetwork(nodes, edges) %>% visOptions(highlightNearest = TRUE)
```

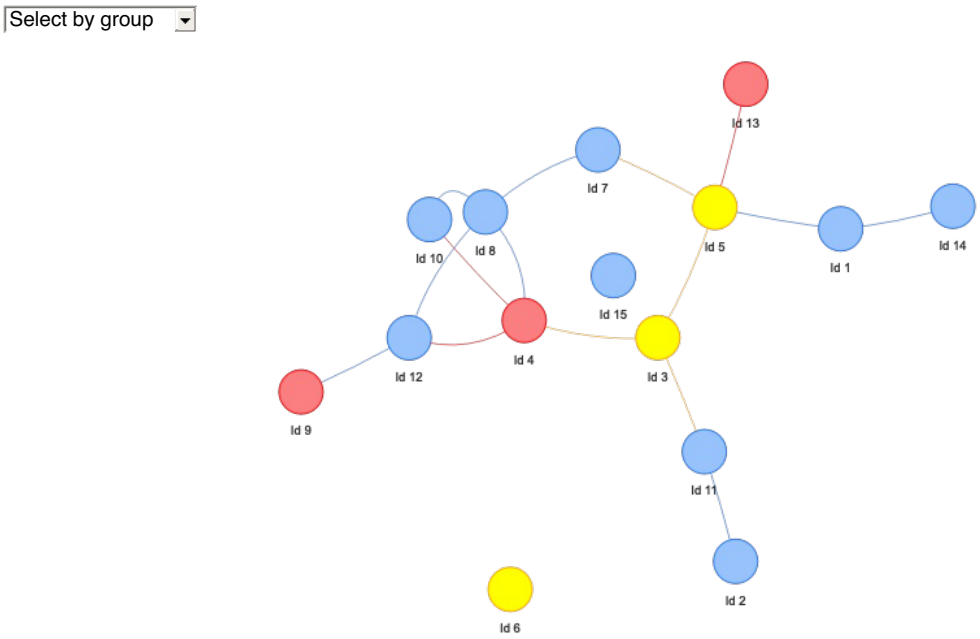



```
visNetwork(nodes, edges) %>% visOptions(highlightNearest = TRUE, nodesIdSelection = TRUE)
```

Select by id




```
visNetwork(nodes, edges) %>% visOptions(selectedBy = "group")
```



Exercice 1.8 (Interactions entre media)**.**

On considère à nouveau le graphe sur les médias

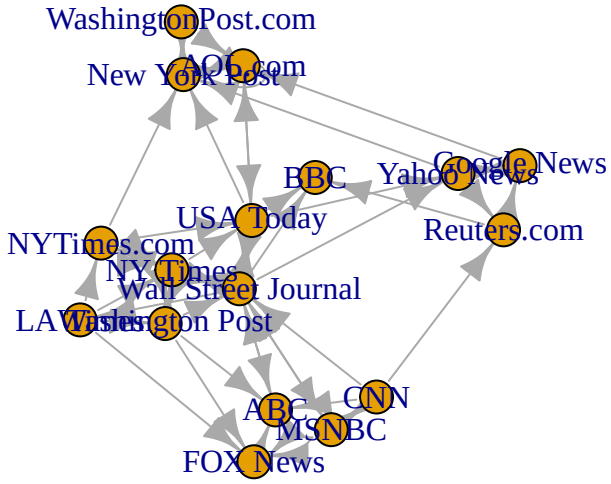
```
nodes <- read.csv("data/Dataset1-Media-Example-NODES.csv", header=T, as.is=T)
links <- read.csv("data/Dataset1-Media-Example-EDGES.csv", header=T, as.is=T)
head(nodes)
   id               media media.type type.label
1 s01            NY Times          1  Newspaper
2 s02     Washington Post          1  Newspaper
3 s03 Wall Street Journal          1  Newspaper
4 s04           USA Today          1  Newspaper
5 s05            LA Times          1  Newspaper
6 s06       New York Post          1  Newspaper
  audience.size
1            20
2            25
3            30
4            32
5            20
6            50
head(links)
  from  to weight      type
1  s01 s02     10 hyperlink
2  s01 s02     12 hyperlink
3  s01 s03     22 hyperlink
4  s01 s04     21 hyperlink
5  s04 s11     22   mention
6  s05 s15     21   mention
```

L'objet `nodes` représente les nœuds du graphe et l'objet `links` les arêtes. On définit l'objet `graphe` avec

```
media <- graph_from_data_frame(d=links, vertices=nodes, directed=T)
V(media)$name <- nodes$media
```

et on peut le visualiser en faisant un `plot` de cet objet

```
plot(media)
```



1. Visualiser ce graphe avec **VisNetwork**. On pourra utiliser la fonction **toVisNetworkData**
2. Ajouter une option qui permette de sélectionner le type de media (Newspaper, TV ou Online).
3. Utiliser une couleur différente pour chaque type de media.
4. Faire des flèches d'épaisseur différente en fonction du poids (weight). On pourra également ajouter l'option **visOptions(highlightNearest = TRUE)**.

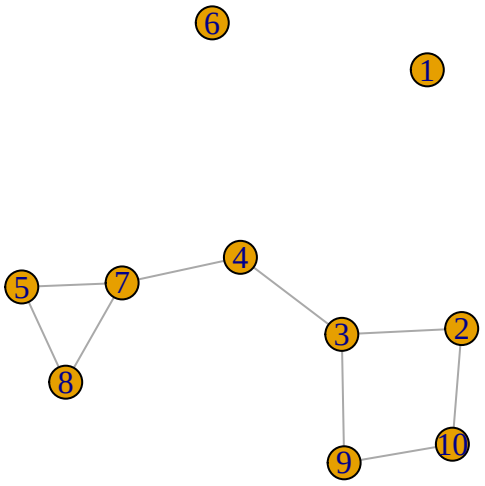
1.4.2 Graphes ggplot avec ggnet

Les fonctions **ggnet** et **ggnet2** du package `GGally` permettent de tracer des **graphes ggplot**. On pourra trouver un descriptif clair à l'url suivante https://briatte.github.io/ggnet/.

```
library(GGally)
```

On construit un premier graphe que l'on visualise avec **igraph**.

```
set.seed(1)
G <- sample_gnp(10,0.2)
plot(G)
```

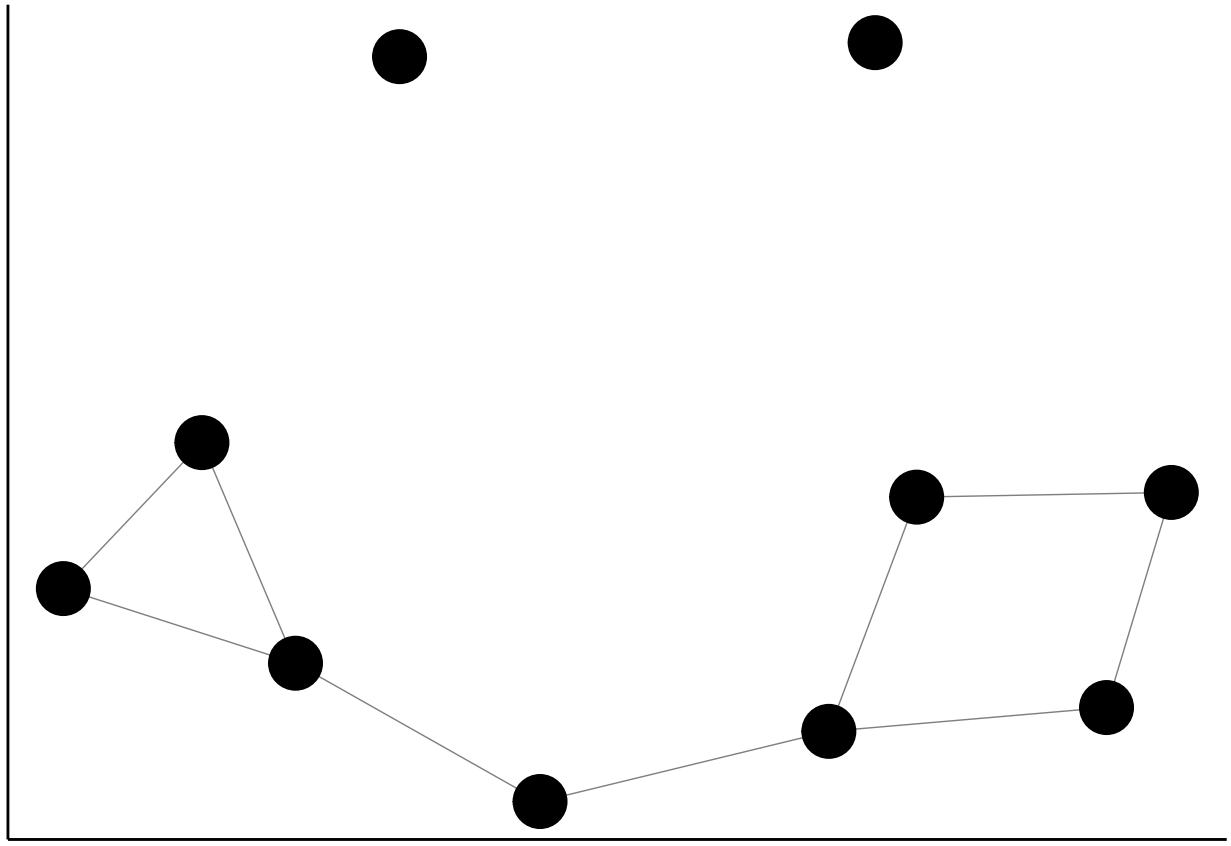


Pour visualiser ce graphe en **ggplot** il faut le transformer en objet **network** :

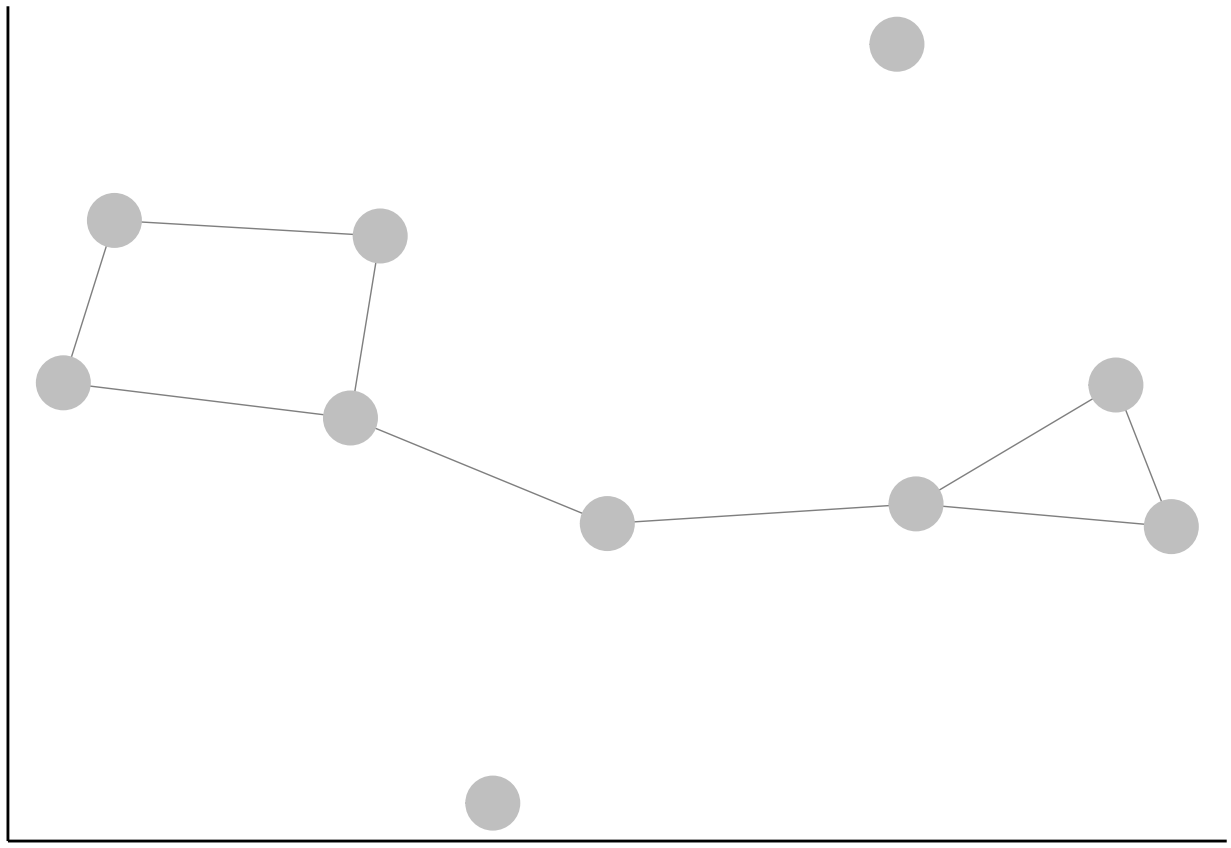
```
net <- igraph::as_data_frame(G) %>% network::as.network()
```

On peut maintenant utiliser les fonctions **ggnet** et **ggnet2**.

```
ggnet(net)
```

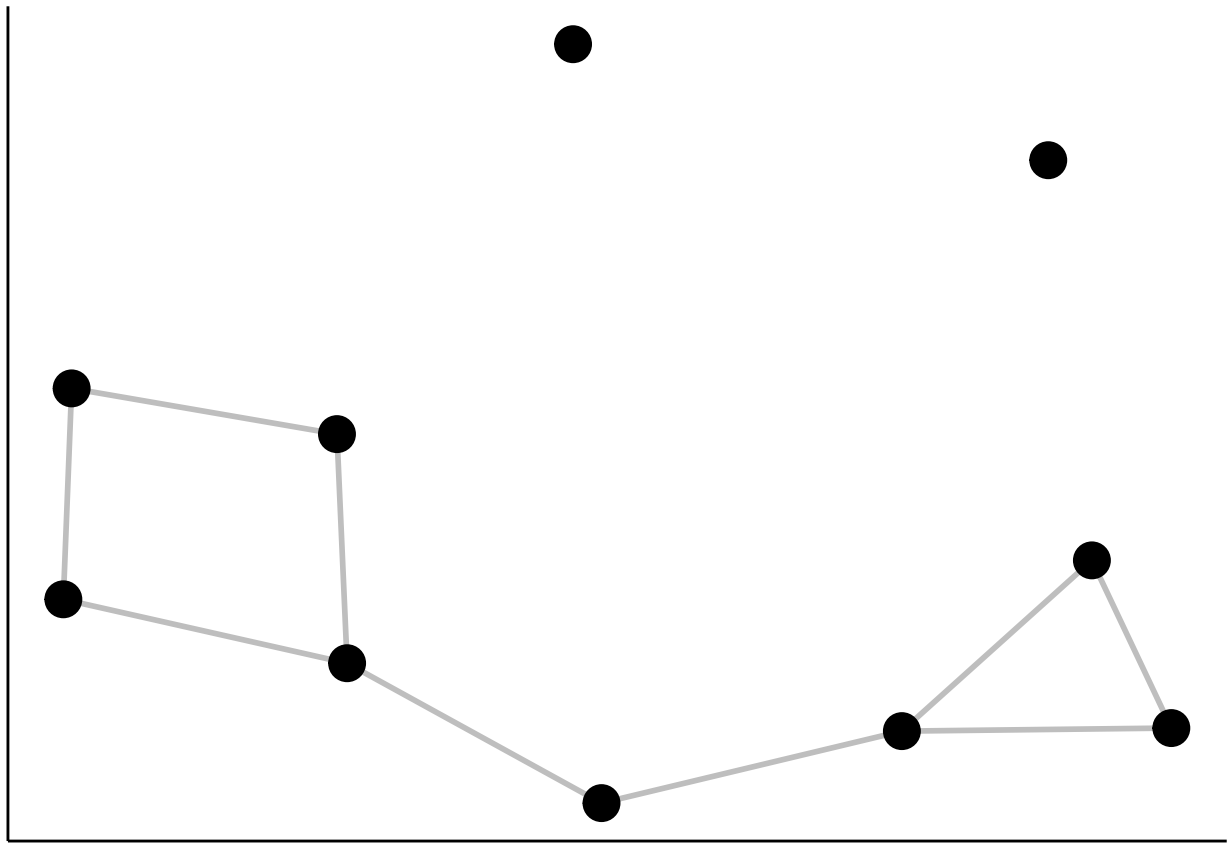


```
ggnet2(net)
```

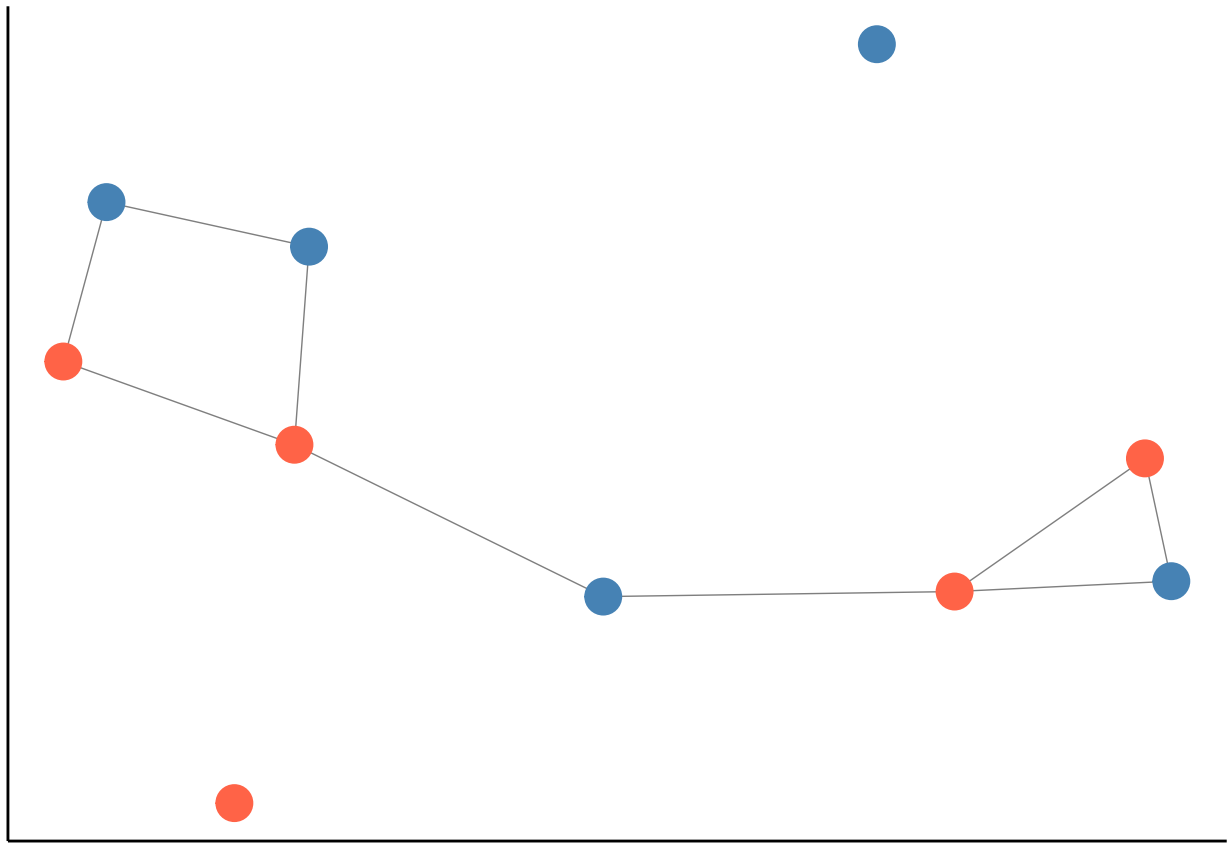


On retrouver bien entendu la plupart des options standards pour visualiser les noeuds et arêtes, par exemple

```
ggnet2(net, node.size = 6, node.color = "black", edge.size = 1, edge.color = "grey")
```



```
ggnet2(net, size = 6, color = rep(c("tomato", "steelblue"), 5))
```



2 Modèles et construction de graphes

2.1 Modèles de graphe

Nous proposons dans cette partie de générer des graphes selon différents modèles de graphes aléatoires présentés en cours.

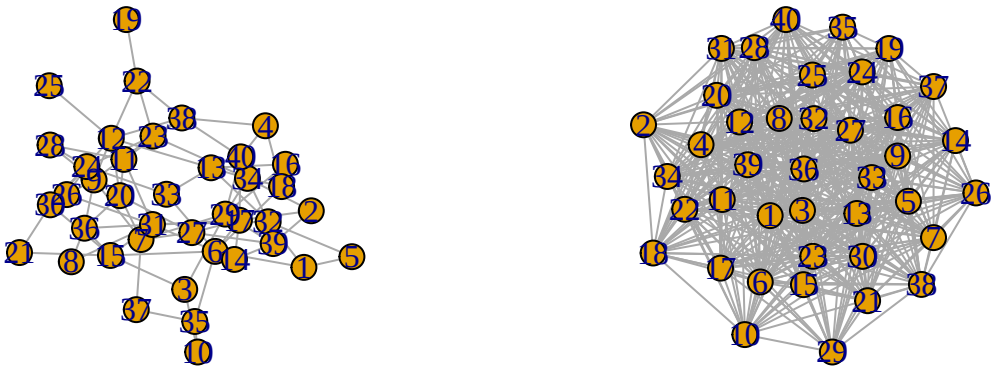
2.1.1 Graphe d'Erdos-Renyi

La fonction **sample_gnp** du package `igraph` permet de simuler un graphe d'**Erdos-Renyi**. On donne comme paramètres

- n le nombre de nœuds
- p la probabilité de connexion entre deux nœuds.

On simule 2 graphes différents : un avec peu de connexions, et un autre très connecté.

```
set.seed(1)
n <- 40
p1 <- 0.1
p2 <- 0.7
G1 <- sample_gnp(n,p1)
G2 <- sample_gnp(n,p2)
par(mfrow=c(1,2))
plot(G1)
plot(G2)
```



On rappelle que dans un graphe d'Erdos-Renyi la distribution du degrés du nœud i est binomiale : $\mathcal{B}(n-1,p)$.
Exercice 2.1 (Distribution des degrés)**.**

A l'aide d'un diagramme en barre, comparer les distributions empiriques des degrés des noeuds (on pourra utiliser **degree.distribution**) à leur distribution théorique binomiale (**dbinom**) pour les deux graphes précédents.
Exercice 2.2 (Les misérables sont-ils des Erdos-Renyi)**.**

On considère le graphe sur **les misérables** où une interaction entre deux personnages est définie par la co-occurrence des ces deux personnages dans un même chapitre.

```
miserab <- read.graph('data/lesmis.gml',format="gml")
```

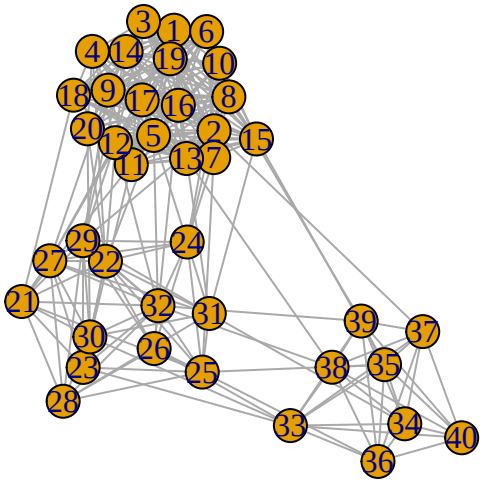
1. Visualiser la distribution des degrés de ce graphe.
2. On souhaite comparer cette distribution à celle d'un graphe d'Erdos-Renyi. Proposer un moyen d'estimer les paramètres (n et p).
3. Comparer la distribution empirique du graphe à celle théorique.

2.1.2 Modèles à blocs stochastiques

La fonction **sample_sbm** du package `igraph` permet de simuler un graphe **SBM**.

```
n <- 40 # nombre de noeuds
Q <- 3 # nombre de clusters
pi <- c(0.5, 0.3, 0.2) # appartenance aux groupes
effectifs <- n*pi

connectivite_matrix <- matrix(c(0.9, 0.1, 0.04,
                                0.1,0.7, 0.05,
                                0.04, 0.05, 0.95),nrow=Q) # matrice de connexion inter/intra groupes
G <- sample_sbm(n, pref.matrix=connectivite_matrix, block.sizes = effectifs)
plot(G)
```



On visualise qu'il s'agit bien d'un graphe avec trois **communautés** ou **groupes**, ce qui est dû aux fortes probabilités sur la diagonale de la matrice de connectivité et aux faibles valeurs de connectivité en dehors la diagonale.

Exercice 2.3 (Clustering avec un modèle **SBM**)**.**

On considère le graphe G construit précédemment.

1. Calculer la matrice d'adjacence du graphe. On pourra utiliser **as_adj**.
2. Les commandes suivantes permettent d'estimer les paramètres d'un graphe **SBM**

```
library("blockmodels")
mysbm <- BM_bernoulli('SBM_sym',A,verbosity=0) # SBM_sym = non dirigé
mysbm$estimate()
```

   Que pouvez-vous dire à propos du nombre de groupes ?
3. À l'aide des sorties présentes dans **mysbm$model_parameters**, récupérer l'estimation de la matrice de connectivité. Comparer aux vraies valeurs.
4. On trouve dans **mysbm$memberships[[3]]$Z** les estimations des probabilités a posteriori d'être dans chaque cluster. Déduire de cette matrice un groupe pour chaque observation.
5. Visualiser les clusters sur le graphe.

Exercice 2.4 (SBM pour le karaté)**.**

A l'aide d'un modèle SBM, identifier des clusters ou communautés sur le graphe **karate**.

2.2 Construire un graphe à partir de données "classiques"

Dans de nombreuses applications on ne dispose pas du graphe, l'utilisateur doit le construire à partir d'un jeu de données standard **individus-variabes** Les méthodes classiques consistent à calculer des distances entre les individus et à mettre une arrête lorsque des individus sont "proches". La notion de proximité est bien entendu à définir, il existe plusieurs possibilités :

— ε-neighborhood graph : on met une arête entre i et j si la distance entre i et j est plus petite qu'un seuil ε ;
— plus proches voisins : on met une arête entre i et j si i est parmi les plus proches voisins de j.

Exercice 2.5 (Plus proches voisins pour les iris)**.**

On reprend le jeu de données **iris** vu en cours, dont on extrait un sous échantillon.

```
data(iris)
set.seed(12345)
donnees <- iris[sample(nrow(iris),30),]
head(donnees)
    Sepal.Length Sepal.Width Petal.Length Petal.Width
142          6.9         3.1          5.1         2.3
51           7.0         3.2          4.7         1.4
58           4.9         2.4          3.3         1.0
93           5.8         2.6          4.0         1.2
75           6.4         2.9          4.3         1.3
96           5.7         3.0          4.2         1.2
       Species
142  virginica
51  versicolor
58  versicolor
93  versicolor
75  versicolor
96  versicolor
```

1. Construire les distances euclidiennes entre individus en ne considèrant que les 4 variables quantitatives. On stockera ces distances dans une matrice et on visualisera cette matrice à l'aide d'un **heatmap**.
2. A l'aide de la fonction **nng** du package `cccd`, construire :
   - un graphe de plus proches voisins à 20 ppv
   - un graphe de plus proches voisins à 2 ppv
   - un graphe de plus proches mutuels voisins à 20 ppv
   - un graphe de plus proches mutuels voisins à 2 ppv

   Ces 4 graphes seront non dirigés.
3. Comparer les nombres d'arêtes de chaque graphe.
4. On considère maintenant le graphe de ppv (non mutuels) à 10 ppv. Ajuster un modèle SBM à 3 groupes sur ce graphe. Comparer les groupes obtenus aux espèces d'iris.

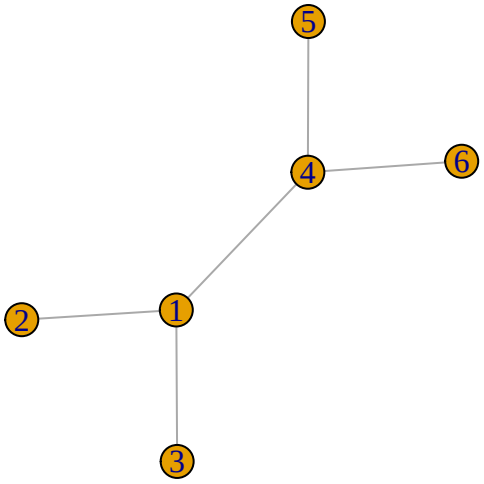
3 Détection de communautés : approche modularité

Une problématique souvent liée aux graphes est la **détection de communautés**. Elle consiste à trouver des groupes de nœuds très liés entre eux. Cette thématique est proche du clustering. Nous présentons dans cette partie les approches liés à la **modularité**. Cette dernière est un critère qui permet de mesurer la performance d'une partition de nœuds dans un graphe, plus la modularité est grande, meilleure est la partition.

Exercice 3.1 (Calculs de modularité)**.**

On considère le graphe suivant

```
G <- make_graph(c(1,2,1,3,1,4,4,5,4,6),directed = FALSE)
plot(G)
```



et les deux partitions des nœuds suivantes.

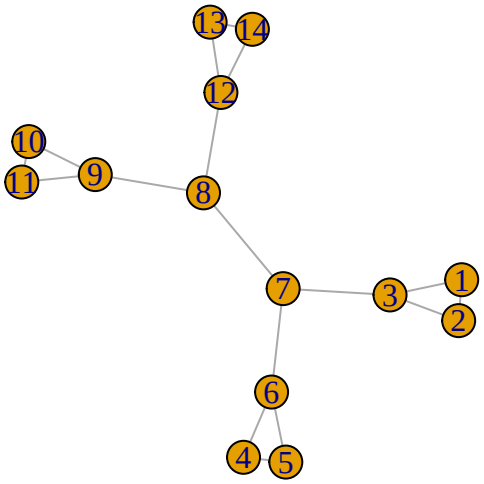
```
cl1 <- c(1,1,1,2,2,2)
cl2 <- c(1,2,1,2,1,1)
```

1. Calculer la **modularité** pour ces deux partitions en utilisant la définition (la formule).
2. Retrouver ces deux valeurs avec la fonction **modularity**.
3. Construire un graphe et proposer une partition avec une modularité élevée et une autre avec une modularité faible.

Exercice 3.2 (Edge betweeness et méthode de Louvain)**.**

On considère le graphe suivant :

```
G1 <- make_full_graph(3)
G2 <- make_full_graph(3)
G3 <- make_full_graph(2)
G4 <- make_full_graph(3)
G5 <- make_full_graph(3)
G <- G1+G2+G3+G4+G5
G <- add.edges(G, c(6,7))
G <- add.edges(G, c(3,7))
G <- add.edges(G, c(8,9))
G <- add.edges(G, c(8,12))
plot(G)
```



1. Calculer l'**edge betweeness** de chaque arête et identifier l'arête qui possède la plus forte valeur.
2. Effectuer le clustering par edge betweeness et visualiser le **dendrogramme**. Identifier la première arête retirée.
3. Représenter les classes sur le graphe.
4. Couper le dendrogramme pour obtenir 3 classes. On pourra utiliser **cutat**.
5. Comparer le résultat avec la méthode de Louvain.
6. Comparer les modularités obtenues.
7. Comparer avec **cluster_optimal**.

Exercice 3.3 (Communautés pour karaté et friends)**.**

Utiliser les techniques basées sur la modularité pour faire de classes sur les données **karate** et **friends**.

```
library(igraphdata)
data(karate)
```

4 Clustering spectral

Le *clustering spectral* est un algorithme de classification non supervisé qui permet de définir des clusters de nœuds sur des graphes ou d'individus pour des données `individus/variables`. L'algorithme est basé sur la décomposition spectrale du Laplacien (normalisé) d'une matrice de similarité, il est résumé ci-dessous :

Entrées :

— tableau de données $n \times p$
— K un noyau
— k le nombre de clusters.

1. Calculer la matrice de **similarités** W sur les données en utilisant le **noyau** K
2. Calculer le **Laplacien normalisé** L_{norm} à partir de W.
3. Calculer les k **premiers vecteurs propres** $u_1, \ldots, u_k$ de L_{norm}. On note U la matrice $n \times k$ qui les contient.
4. Calculer la matrice T en **normalisant les lignes** de U : $t_{ij} = u_{ij}/(\sum_\ell u_{i\ell}^2)^{1/2}$.
5. Faire un k-**means** avec les points $y_i, i = 1, \ldots, n$ (i-ème ligne de T) $\Longrightarrow A_1, \ldots, A_k$.

Sortie : clusters $C_1, \ldots, C_k$ avec

$$C_j = \{i | y_i \in A_j\}.$$

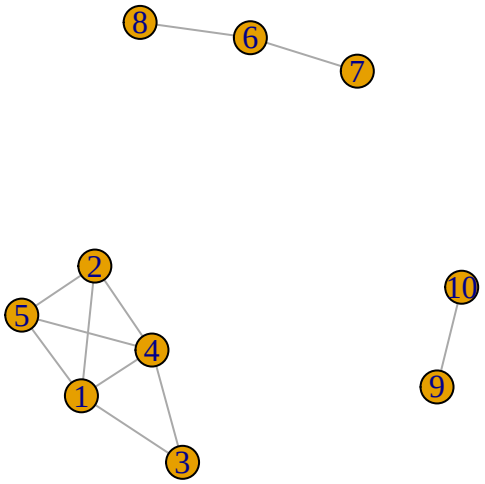
L'objet de ce chapitre est de travailler sur cet algorithme en le programmant, puis en utilisant la fonction **specc** du package `kernlab`.

4.1 Clustering spectral sur 1 graphe à 3 composantes connexes

On crée tout d'abord un graphe avec trois composantes connexes : on utilise la commande **sample_gnp()** qui permet de créer un graphe selon le modèle d'Erdos-Renyi.

```
set.seed(1)
n1 <- 5
n2 <- 3
n3 <- 2
n <- n1+n2+n3
# il faut prendre des grandes valeurs de p sinon on risque d'avoir des sous-graphes non connexes
p1 <- 0.85
p2 <- 0.75
p3 <- 0.7
```

```
G1 <- sample_gnp(n1,p1)
G2 <- sample_gnp(n2,p2)
G3 <- sample_gnp(n3,p3)
G <- G1 + G2 + G3 # il cree un graphe avec ces 3 sous-graphes
plot(G)
```



On vérifie le nombre de composantes connexes

```
components(G)$no
[1] 3
```

Exercice 4.1 (Laplacien non normalisé)**.**

1. Calculer la matrice d'adjacence de **G** et en déduire le Laplacien non normalisé.
2. Retrouver ce Laplacien avec la fonction **laplacian_matrix**.
3. Calculer les valeurs propres et représenter les sur un graphe. Que remarquez-vous ?
4. Obtenir les trois vecteurs propres associés à la valeur propre nulle. Commenter.
5. Terminer l'algorithme de clustering spectral avec l'étape de k means.
6. Visualiser les clusters.

Exercice 4.2 (Laplacien normalisé)**.**

Refaire le même travail en utilisant le laplacien normalisé. On n'oubliera pas d'ajouter l'étape de normalisation en utilisant par exemple la fonction suivante :

```
normalize <- function(x){
  return(x/sqrt(sum(x^2)))
}
```

4.2 Programmer le clustering spectral pour un graphe

Exercice 4.3 (Construction de l'algorithme)**.**

Créer une fonction **R** qui admet en entrée :

— un graphe
— une valeur de K (un entier positif)

et qui renvoie les groupes pour le clustering spectral à K groupes ainsi que le graphe des valeurs propres (en **ggplot** si possible).

Igraph possède une fonction permettant de faire directement le spectral clustering : **embed_laplacian_matrix**. Mais en argument, il faut lui donner le nombre K de clusters souhaité. En pratique, on ne connait pas K, et une façon de le trouver est de regarder le trou spectral dans le graphe des valeurs propres.

La fonction **embed_laplacian_matrix** s'utilise ainsi :

```
res2 <- embed_laplacian_matrix(G,8,which="sa",scaled="FALSE",degmode = "all")
res2$D
[1] 0 0 0 1 2 2 3 4
```

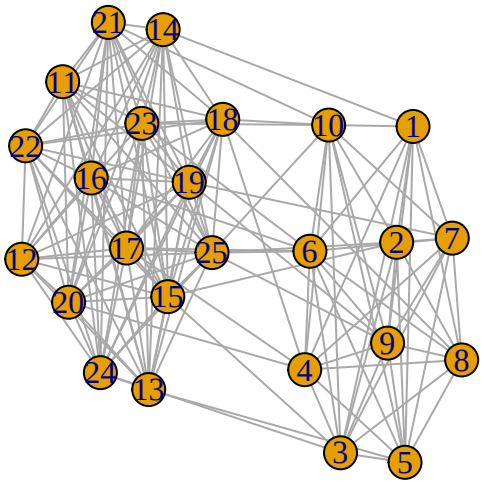
On peut faire du clustering spectral à 3 groupes avec :

```
res3 <- embed_laplacian_matrix(G,3,which="sa",scaled="FALSE",degmode = "all")
res_spectral <- kmeans(res3$X, centers = 3, nstart = 100)
res_spectral$cluster
 [1] 1 1 1 1 1 3 3 3 2 2
```

Exercice 4.4 (Graphe avec deux communautés faiblement connectées entre elles)**.**

On considère le graphe suivant obtenu selon un modèle **SBM** :

```
set.seed(1234)
n <- 25# nombre de noeuds
Q <- 2 # nombre de cluster clusters
pi <- c(0.4, 0.6) # taille des groupes
effectifs <- n*pi
connectivite_matrix <- matrix(c(0.9, 0.15,
                                0.15, 0.95),nrow=Q) # matrice de connexion
G <- sample_sbm(n, pref.matrix=connectivite_matrix, block.sizes = effectifs)
plot(G)
```



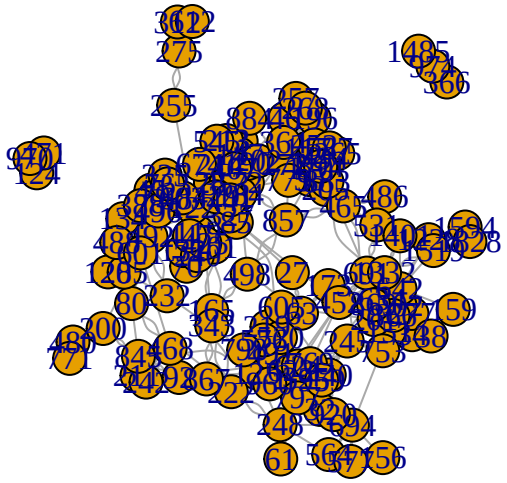
Effectuer le clustering spectral sur ce graphe, on essaiera notamment de choisir le nombre de groupes.

4.3 Exemple sur des graphes "réels"

Exercice 4.5 (Clustering spectral sur deux graphes)**.**

1. On considère le graphe **friends** disponible ici

```
friends <- read.table(file="data/Friendship-network_data_2013.csv")
G.friends <- graph_from_data_frame(friends,directed=F)  # non dirige
plot(G.friends)
```



Appliquer le clustering spectral à ce graphe. On pourra comparer la classification obtenue avec celle de la méthode de Louvain en utilisant la fonction **compare**.

2. Faire de même avec le graphe **karate**.

```
library(igraphdata)
data(karate)
```

4.4 Clustering spectral : cas général

Nous avons étudié jusqu'ici l'algorithme du clustering spectral pour trouver des clusters de nœuds (ou communautés) dans les graphes. On remarque néanmoins que l'algorithme ne repose pas sur le graphe en lui même, mais uniquement sur une matrice d'adjacence (ou similarité) issue de ce graphe. Il est par conséquent possible d'utiliser cet algorithme pour des données standards (tableaux `individus-variables`), à partir du moment où on peut calculer une matrice de similarité à partir de ces données. Il est également possible d'utiliser des **noyaux** pour définir cette similarité. La fonction **specc** de `kernlab` permet de faire un tel clustering.

Exercice 4.6 (Clustering spectral pour des psirales)**.**

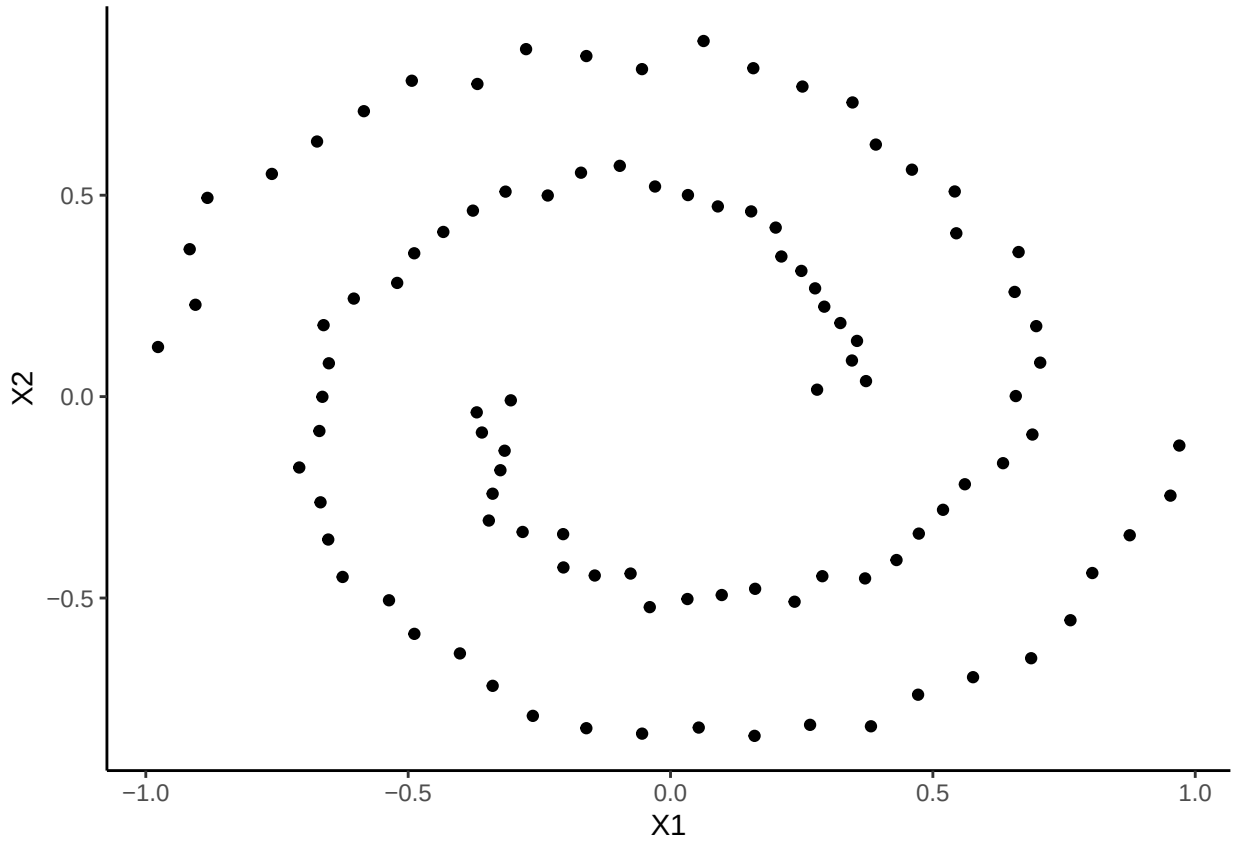
On considère les données spirales

```
set.seed(111)
library(mlbench)
n <- 100
simu <- mlbench.spirals(100,1,0.025)
names(simu)
[1] "x"       "classes"
```

```
data <- simu$x
head(data)
           [,1]        [,2]
[1,]  0.5609898 -0.21756239
[2,]  0.2793522  0.01718273
[3,]  0.3725821  0.03849122
[4,]  0.3457879  0.08963081
[5,]  0.1577921  0.81528541
[6,] -0.1603697  0.84547763
```

et on les visualise.

```
df <- data.frame(simu$x)
ggplot(df)+aes(x=X1,y=X2)+geom_point()
```



Appliquer les algorithmes suivants pour tenter de visualiser les deux groupes :

— clustering spectral avec noyau linéaire
— clustering spectral avec noyau polynomial de degree 2
— clustering spectral avec noyau radial
— k-means
— CAH avec single linkage
— CAH avec lien de Ward